**Конспект лекцій з дисципліни**

**«Програмування систем збору і аналізу даних»**

**Укладач: Тищенко К.В.**

**ЗМІСТ**

**ТЕМА 1. ЗАГАЛЬНІ ВІДОМОСТІ** **ПРО МІКРОКОНТРОЛЕРИ**

* 1. **Структура і принцип** **роботи контролера**

**1.1.1 Центральний процесор**

**1.1.2 Оперативна пам'ять і пам'ять програм**

**1.1.3 Зовнішні пристрої**

**1.2** **Порівняння технологій RISC і CISC**

**1.2.1 Технологія CISC**

**1.2.2 Технологія RISC**

**1.3 Програмування мікроконтролерів**

**1.3.1 Поняття програми**

**1.3.2 Програмування на мові С**

**ТЕМА 2 ПЛАТФОРМА ARDUINO**

**2.1. Апаратна частина**

**2.2 Різновиди плат Arduino. клони, оригінали та сумісність**

**2.2.1 Оригінальні плати.**

**2.2.2 Клони**

**2.2.3 Сумісні**

**2.2.4 Ардуіно-подібні**

**2.3 Підготовка до роботи з Arduino**

**2.3.1 Установка драйвера**

**2.3.2 Установка програмного забезпечення Arduino**

**2.3.3 Середовище розробки Arduino**

**ТЕМА 3. ОСНОВИ ПРОГРАМУВАННЯ ARDUINO**

**3.1 Біти і байти**

**3.2 Базова структура програми**

**3.3 Послідовне** **виконання програми**

**3.4 Переривання виконання програми**

**3.5 Структура програми Arduino**

**3.6 Команди Arduino і їх застосування**

**3.7 Типи даних**

**3.8 Оператори**

**3.9 Керуючі конструкції**

**3.10 Цикли**

**3.11Функції та підпрограми**

**РОЗДІЛ 4. ПРИКЛАДНЕ ПРОРАМУВАННЯ В СЕРЕДОВИЩІ ARDUINO**

**4.1 Послідовний інтерфейс введення / виведення**

**4.1.1 Функції роботи з послідовним інтерфейсом**

**4.1.2 Як функціонує послідовний інтерфейс**

**4.1.3 Програмна емуляція UART**

**4.2 Конфігурація входу / виходу та настройки порту**

**4.3 Зчитування стану кнопки**

**4.3.1 Кнопка з підтягуючим резистором**

**4.3.2 Кнопка з узгоджуючим резистором**

**4.4 Введення аналогових даних і АЦП**

**4.5 Аналоговий вихід. ШІМ**

**4.6 Деякі спеціальні функції**

**4.7 Вимірювання часових інтервалів**

**ТЕМА 5 ПРОТОКОЛИ ЗВ’ЯЗКУ**

**5.1 Використання протоколу I2C**

**5.1.1 Описінтерфейсу I2C**

**5.1.2 Реалізація I2Cв Arduino**

**5.2 Використання протоколу SPI**

**5.2.1 Загальні відомості про протокол SPI**

**5.2.2 Підключення пристроїв SPI**

**5.2.3 Конфігурація інтерфейсу SPI**

**5.2.4 Протокол передачі даних SPI**

**5.2.5 Порівняння SPI і I2C**

**5.3 Енергонезалежна пам'ять EEPROM**

**5.4 Використання переривань в Arduino**

**5.4.1 Поняття переривання**

**5.4.2 Переривання по таймеру**

**5.4.3 Таймери на Arduino**

**5.4.4 Установка Таймера 2**

**5.4.5 Завантаження мікроконтролера перериваннями**

**5.4.6 Вимірювання завантаження переривань**

**5.4.7 ISR Таймера2**

**5.4.8 Головна програма. Функція Setup ()**

**5.4.9 Головна програма. функція Loop ()**

**ТЕМА 1. ЗАГАЛЬНІ ВІДОМОСТІ** **ПРО МІКРОКОНТРОЛЕРИ**

Перед тим як почати працювати з апаратної обчислювальної платформою Arduino, важливо отримати загальні відомості про мікроконтролери. Мікроконтролери застосовуються, перш за все, для автоматизації в метрології, техніці управління і автоматичного регулювання. Перевага мікроконтролерів у тому, що з їх допомогою можна ефективно і з малими витратами вимірювати і інтерпретувати фізичні величини, щоб потім приймати необхідні рішення і виконувати необхідні дії.

Область можливих застосувань мікроконтролерів надзвичайно широка: від домогосподарства (наприклад, для управління теплицею або освітленням) до промислового виробництва, де можуть обслуговуватися і експлуатуватися комплексні пристрої, керовані системами мікроконтролерів. На рис. 1.1 наведено типовий приклад обробки даних для управління зрошувальної установки теплиці. Контролер фіксує дані про температуру навколишнього середовища, вологості ґрунту, отримані від датчиків. Результати вимірювання далі піддаються логічній обробці в мікроконтролері, після чого формуються сигнали управління насосом для поливу.

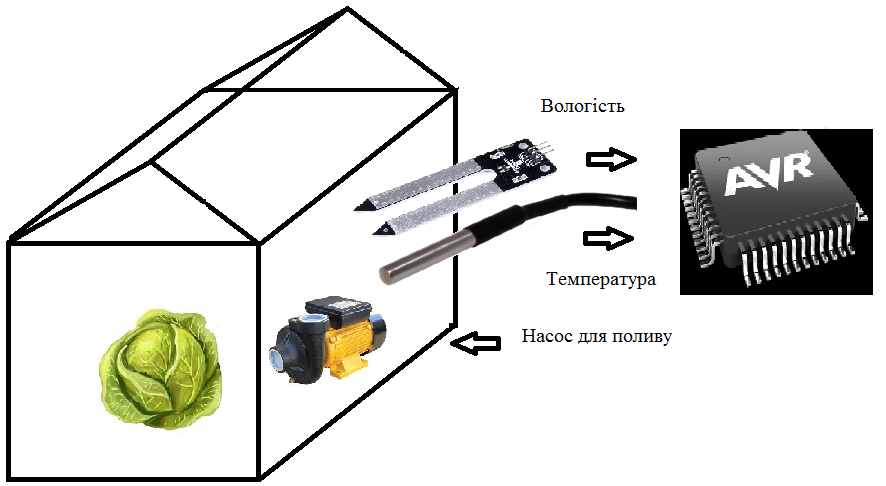


Рисунок 1.1 – Система управліненя мікрокліматом теплиці

**1.1 Структура і принцип** **роботи контролера**

Контролер представляє собою, по суті, мікрокомпьютер і містить всі властиві йому основні модулі (рис. 1.2). Стандартні блоки кожного мікроконтролера – це центральний процесор (CPU), оперативна пам'ять (RAM), а також пам'ять програм (Flаsh-пам'ять) і зовнішні пристрої.

Генератор

Flash пам’ять

ОЗП

IRQ (обробка переривань)

Лічильник

Зовнішні пристрої (I/O)

Процесор

Внутрішня шина

Рисунок. 1.2. – Спрощена структура мікроконтролера

* + 1. **Центральний процесор**

Основний функціональний пристрій мікроконтролера – центральний процесор (CPU - Central Processing Unit). Його можна порівняти з "мозком" мікроконтролера. Сигнали в ньому представлені в цифровій формі і над ними виконуються арифметичні і логічні операції.

**1.1.2 Оперативна пам'ять і пам'ять програм**

Оперативна пам'ять і пам'ять програм традиційно розглядаються окремо. Програма користувача, яку ми самі писали, зберігається в незалежній Flаsh- пам'яті програм. В залежності від типу контролера пам'ять програм може займати об'єм від декількох кілобайт до мегабайт. Крім того, в деяких обчислювальних системах можна збільшити пам'ять програм, підключаючи зовнішні Flаsh-накопичувачі.

Оперативна пам'ять служить для тимчасового зберігання різних проміжних даних. Тут зберігаються і результати обчислень, отримані під час виконання програми. Призначення оперативної пам'яті ОЗУ (RАМ - Random Access Memory) – можливість швидкого звернення до обмеженої кількості даних. Її об’єм, як правило, значно менший, а швидкодія набагато більшае, ніж Flаsh-пам'яті. Значення записуються і зберігаються в ОЗП під час виконання і енергозалежна на відміну від Flаsh-пам'яті, тобто після перезавантаження контролера вміст ОЗП повністю стирається. На рис. 1.3 зображена структура ОЗП мікроконтролера ATmega328P.

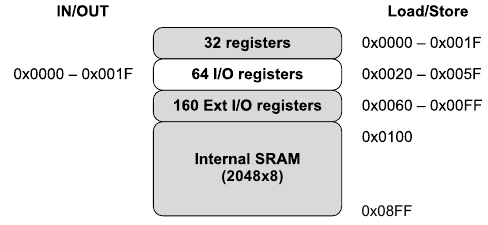


Рисунок. 1.3. – RАМ-пам'ять мікроконтролера ATmega328P

**1.1.3 Зовнішні пристрої**

Зовнішніми (периферійними) пристроями часто називають всі компоненти мікроконтролера, крім центрального процесора і пам'яті. Зокрема, до них відносяться зовнішні інтерфейси, наприклад, цифрові входи і виходи (lnput / Output – скорочено I/O). Більшість мікроконтролерів забезпечені також різними цифровими і аналоговими входами і виходами.

**1.2** **Порівняння технологій RISC і CISC**

Розгляд RISC- і СISС-технологій – це вже більш глибокий погляд на цифрову і мікроконтролерну техніку. Відразу слід зауважити, що контролери АVR для Arduino базуються на технології RISC.

**1.2.1 Технологія CISC**

При технології CISC до ОЗП завантажується і програма, і дані. Кажуть також про те, що код програми і дані ділять між собою одну і ту ж область пам'яті. Це мало сенс, зокрема, в перших обчислювальних системах, оскільки оперативна пам'ять була дорогою.

Для мікроконтролера значно важливішою відмітною ознакою є структура команд. Комп'ютер з СISС-технологією має в своєму розпорядженні великий асортимент дуже вузькоспеціалізованих команд. У цифровій техніці команда – це послідовність певних байт. Один байт може приймати 256 різних станів. Щоб реалізувати більше 256 різних команд, потрібні додаткові байти. Таким чином, спеціальна команда займає багато місця, наприклад п'ять, байт. Завантаження цієї команди триває довше, ніж короткої однобайтовї команди.

**1.2.2 Технологія RISC**

Було встановлено, що в СISС-комп'ютерах, як правило, близько 90% вихідного коду програм складається тільки з 30 різних команд. Так виникла думка реалізувати в центральному процесорі систему обмеженого числа коротких і швидко виконуваних команд. Таким чином, в RISС-мікроконтролерах команди, зазвичай, складаються не більше ніж з 1-2 байт. Довгу спеціальну команду доводиться складати їх з декількох коротких. Щоб досягти рівної продуктивності з СISС-комп'ютерами, більшість RISС-комп'ютерів мають велике число регістрів. Регістр – це вбудована в центральний процесора надшвидкісна пам'ять. Ще одна відмінна ознака RISС-систем – чітке фізичне і логічне розділення між областями пам'яті програм і даних.

Тож можна сказати, що СISС-комп'ютер має безліч спеціальних команд, які займають великий обсяг пам'яті і вимагають, як правило, тривалого часу виконання. Команди RISС-комп'ютера споживають менше пам'яті і виконуються набагато швидше. Тим неменш, недолік RISС-технології полягає в тому, що спеціальні команди доводиться замінювати ланцюжками з декількох основних команд. Таким чином, і CISC- і RISС-технологія маю свої переваги і недоліки. Слід зазначити, що не існує ні повністю RISC-, ні повно стю СISС-систем.

**1.3 Програмування мікроконтролерів**

Ступінь інтеграції мікропроцесорів і мікроконтролерів все більше зростає, і вони все ширше проникають в прикладні галузі метрології, техніки управління, автоматичного регулювання. Навіть в звичайному житті мікроконтролери стають дедалі популярнішими. Так відбувається, з одного боку, у зв’язку з тим, що сьогодні складні аналогові схеми замінюються більш простими цифровими мікроконтролерами. Але вирішальна перевага мікроконтролерів – це неперевершене співвідношення ціна / продуктивність.

**1.3.1 Поняття програми**

Програма – це опис процесу обробки інформації. При виконанні програми розраховується сукупність вихідних значень виходячи із сукупності змінних або постійних вхідних значень. Мета виконання програми – збір даних або отримання відгуку на вхідні значення. Наряду із власне обчисленнями програма може містити команди для доступу до апаратних засобів комп'ютера і для управління ходом виконання алгоритму. Програма складається з декількох рядків так званого початкового тексту. При цьому кожен рядок містить один або кілька арифметичних або управляючих операторів. Не тільки самі команди, а і послідовність їх виконання істотно впливає на результат обробки інформації. Виконання відповідних операцій відбувається послідовно (по черзі). Впорядковану певним чином послідовність інструкцій програми називають також алгоритмом.

**1.3.2 Програмування на мові С**

Мова програмування С (ANSI С) достатньо проста для вивчення. С – це мова програмування високого рівня, яку створив Денніс Річі (Dennis Ritchie) на початку 1970-х років в Bell Laboratories для операційної системи UNIX. З тих пір ця мова дуже широко поширена. Області застосування мови С досить різноманітні. Вона використовується, наприклад, в системному і прикладному програмуванні. Основні модулі всіх систем UNIX і ядро багатьох операційних систем запрограмовані на мові С.

Численні інші мови, наприклад С ++, Objective-C, С #, Java, РНР і Perl орієнтуються на синтаксис і властивості мови С. Вивчення цієї мови програмування дуже вигідне, оскільки у подальшому легше освоїти багато систем мікроконтролерів. Майже для всіх мікроконтролерів існує безкоштовний компілятор С, пропонований виробником мікроконтролера. Компілятор С від Arduino дещо простіший, ніж професійні С-компілятори, але досить ефективний. З компілятором Arduino не потрібно піклуватися про програмування складних апаратних засобів, оскільки в середовищі розробки є відповідні вбудовані команди.

**ТЕМА 2 ПЛАТФОРМА ARDUINO**

Arduino – апаратна обчислювальна платформа, основними компонентами якої є плата вводу / виводу на базі мікроконтролера AVR і середовище розробки на мові Arduino (C/C ++).

**2.1. Апаратна частина**

Апаратні засоби Arduino включають популярні та доступні комплектуючі, що випускаються серійно, тому принцип роботи системи зрозумілий, настройка схеми під вимоги розробника проста і забезпечена можливість подальшої модифікації.

Плата Arduino складається з мікроконтролера Atmel AVR (ATmega328 і ATmega168 в нових версіях і ATmega8 в старих) та елементної обв'язки для програмування та інтеграції з іншими схемами. На кожній платі обов'язково присутні лінійний стабілізатор напруги 5 В (у деяких моделей 3,3 В) і кварцовий резонатор 16 МГц. У мікроконтролер попередньо прошитий завантажувач, тому зовнішній програматор не потрібен. На концептуальному рівні, всі плати програмуються через інтерфейс RS-232 (послідовне з'єднання), але реалізація цього способу відрізняється від версії до версії. Плата Serial Arduino містить просту інвертуючу схему для конвертації рівнів сигналів RS-232 в рівні ТТЛ, і навпаки. Актуальні насьогодні плати, на зразок Diecimila, програмуються через USB, що здійснюється через мікросхему конвертера USB-to-serial, наприклад FTDI FT232, або Prolific PL2303. У деяких варіантах, таких як Arduino Mini або неофіційнї Boarduino, для програмування потрібне підключення окремої плати або кабеля USB-to-serial.

Плати Arduino дозволяють використовувати більшу частину I/O виводів мікроконтролера в зовнішніх схемах. Наприклад, в платі Diecimila доступно 14 цифрових вводів / виводів (рівні «LOW» – 0В і «HIGH» –5В), 6 з яких можуть видавати ШІМ сигнал, і 6 аналогових входів (0-5В). Ці виводи доступні на верхній частині плати через 0,1 дюймові роз'єми типу «мама». На ринку доступні кілька зовнішніх плат розширення, відомих як «shields».

Назва «Arduino» (і похідні від нього) є торговою маркою для офіційного продукту і не може використовуватися для похідних робіт без дозволу. В офіційному документі, про використання назви Arduino, підкреслюється, що проект відкритий для всіх бажаючих працювати над офіційним продуктом. Результатом захисту назви стало відгалуження від Arduino групи користувачів, що призвело до випуску еквівалентної плати, названої Freeduino, назва якої не є торговою маркою і може використовуватися в будь-яких цілях.

**2.2 Різновиди плат Arduino. клони, оригінали та сумісність**

Періодично команда розробників Arduino випускає нові плати, також світовою спільнотою розроблено та клоновано ще велику кількість плат, спробуємо розібратися, що із себе представляють ці розробки.

Для початку слід зазначити, що всі «розміри» і різновиди Ардуіно-плат абсолютно сумісні один з одним – якщо вас зацікавив проект на Ardino Nano – то ви з легкістю зможете реалізувати його на Ardino Uno або Ardino Mega, причому ні в коді ні в схемі переробляти нічого не доведеться. Можна і навпаки, наприклад, з Ardino Mega на Ardino Mini – лише б виводів та пам'яті вистачило. Так само ніякої різниці немає у виборі конкретної плати всередині розмірного ряду – можна взяти проект для ArduinoDiecimila і спокійно портувати його на UNO і навпаки. Тим більше немає ніякої принципової різниці, хто зробив цю плату і як вона називається, основою буде все той же мікроконтролер ATMega.

**2.2.1 Оригінальні плати.**

Розробники випускають плату в декількох основних форм-факторах:

* Ardino xxx – стандартний розмір, 20 входів/виходів, повна сумісність з усіма Шілд.
* ArdinoMega xxx – збільшений розмір, 70 входів/виходів, сумісність не з усіма Шілд.
* ArdinoNano xxx – зменшений розмір, 22 входів/виходів, не сумісна з Шілд.
* ArdinoMini ххх – ще менший розмір, 20 входів/виходів, не сумісна з Шілд, не має USB.

**Ardino** xxx

Стандартний і найпоширеніший розмір. Коли говорять «Ардуіно» («звичайна Ардуіно») – зазвичай мають науваза саме такі плати.

Найперші плати були в цьому форм-факторі, відповідно саме він пережив найбільше реінкарнацій (USB-версії в хронологічному порядку виходу):

Extreme, NG, Diecimila, Duemilanove, Uno, Leonardo.

Зараз на офіційному сайті доступні для покупки тільки Leonardo і Uno, проте в інтернеті можна придбати велику кількість різноманітних плат у цьому форм-факторі, оскільки він найбільш підходить середньому користувачеві. Всі ці плати мають однакову кількість входів / виходів, зібраних на однакових роз'ємах (для підключення переферії і Шілд), програмуються по USB, і базуються на мікроконтролері ATMega. На ранніх версіях був встановлений ATMega8, потім ATMega168 , тепер ATMega328.

На ATMega8 тільки 3 ШІМ виходи, 8Кб під скетч 1Кб оперативної пам’яті, але для багатьох додатків цього вистачає. У ATMega168 вже 6 ШІМ каналів і 16Кб під прошивку, а у 328-й 32Кб під програми і 2Кб ОЗП. До речі не вся флеш-пам'ять доступна користувачу, частину її займає бутлоадер.

На всіх платах до UNO стояв чіп-перетворювач USB-UART FT232 , що дозволяє під’єднувати плату прямо в USB і програмувати без програматора. При підключенні в системі з'являвся віртуальний COM-порт, який і використовується середовищем розробки Ардуіно для програмування.

На Arduino UNO (рис. 2.1 а) вирішили замінити перетворювача USB-UART, на мікроконтролер Atmega8U2 (в більш пізніх ревізіях 16U2 ) – в нього залита спеціальна прошивка, що робить рівно те ж що і FT232. Це дало деякі переваги, наприклад, піднялася швидкість прошивки – тепер замість ~ 10 секунд поотрібно чекати ~ 3c, а головне, в цей МК-конвертор можна записати свою прошивку, і перетворити Ардуіно в мишку, клавіатуру або міді пристрій.

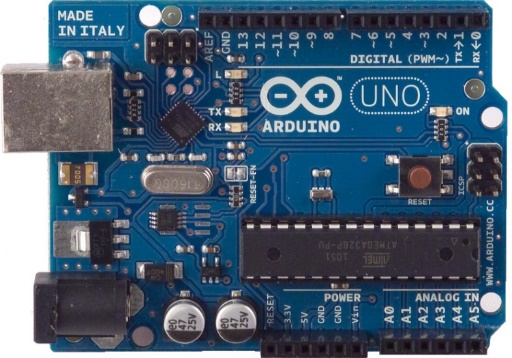
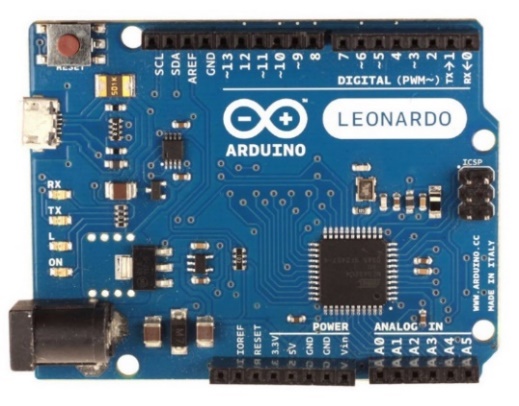
Також, тут ще варто зробити застереження – ця Atmega8U2 / 16U2 насправді робить не рівно те ж, що FT232, вона не реалізує дуже зручної можливості – BitBang, тому перетворити плату в програматор вже не вийде.

В UNO з'явилися на «верхньому лівому» коннекторі SDA і SCL – піни інтерфейсу I2C, але вони дублюють SDA і SCL на аналогових входах 4 і 5, і функціонал це не розширює. На даний момент - старі Шілд підходять до UNO, нові Шілд (яких поки вкрай мало), повністю сумісні зі старими платами хоча і загрожують «уткнутися» в неї новими пінами – їх, можливо, доведеться підігнути або відкусити.

**Leonardo** (рис. 2.1 б)

Ця плата дійсно крок вперед – все розміщено на одному чіпі, USB незалежний ні від UART ні взагалі від яких би то не було пінів. Плата побудована на ATmega32u4 і в порівнянні з попередніми моделями має більші можливості. На 0,5кБ збільшилася ОЗП, ШІМ-виходів стало на 1 більше, аналогових входів стало 12 (6 – де у всіх Ардуіно, нові +6 розкидані по цифровим пінам) і, як уже говорилося, розділені USB і UART. Так само підтримуються, не тільки віртуальний ком порт, але і миша і клавіатура , набагато простіше ніж це реалізовано в UNO.

Правда «крок вперед» вийшов з деякими нюансами – бутлоадер тепер займає 4Кб, а ще в будь-який скетч для Леонардо додається підтримка USB. Найпростіший скетч blink для Duemilanove / UNO займе 1084 байт, а для Leonardo - 4858 байт. Фізично Леонардо має ту ж карту розведення що і UNO, тому він сумісний зі старими Шілд.

а б

Рисунок 2.1. – зовнішній вигляд плат Arduino UNO (а) та Leonardo (б)

**ArdinoMega** xxx

Серія збільшених плат (за розміром і характеристиками) представлена моделями (в хронологічному порядку): Mega, Mega2560 (рис. 2.2) і Arduino ADK.

В плати успішно встромляються майже всі Шілд, але через різне (з «звичайними» Ардуіно) розташування виводів SPI-інтерфейсу , Шілд використовують його з цифрових пинів 11,12,13 будуть не сумісні. Приклад – старий Ethernet Шилд . На новому SPI береться зі стандартної вилки ISP і все відмінно працює і на «мегах», і на «звичайних» ардуіно.

На платах велика кількість виводів:

54 цифрових, з них 15 – із ШІМ,

16 – аналогових входів,

Пам’ять:

128(Mega) / 256кб (Mega2560) – флеш,

8Кб оперативної пам’яті,

4Кб EEPROM

4 фізичних UART

«Мега» побудована на ATmega1280 , а «2560» і «ADK» на ATmega2560 , тому плати розрізняються об’ємом пам'яті, до того ж у нових – 2560 і ADK – USB-частина виконана на ATmega8U2 (на більш пізніх ревізіях 2560 – на ATmega16U2), так само, як і в УНО.

ADK також має USB-host, від якого очікується велика дружба з Android-телефонами.

**ArdinoNano**

Маленька з mini-USB. Шілд до неї не підходять, але сама вона зручно розміщується в макетній платі. У ранніх версіях, як і в UNO використовувалась ATmega168, зараз 328. У якості USB-UART мосту стоять перетворювач FT232.

**ArdinoMini**

Ще менша плата. (тут якийсь історичний ляп – Ардуіно міні, чомусь значно менше Ардуіно нано ) Пережила кілька версій – мають незначні відмінності в призначенні деяких виводів. З Шілд не сумісна, але зручна для вбудовування в закінчені пристрої – нічого зайвого. На міні немає USB – програмується вона за допомогою перехідника USB-Serial ( наприклад на базі тієї ж FT232). Є варіанти плати, що працюють на 3,3 В і 8МГц, що зручно для використання у поєднанні з пристроями мобільного стандарту живлення (3,3 В).



Рисунок 2.2. – Зовнішній вигляд плати Arduino Mega2560

Проект Ардуіно - повністю відкритий (доступна вся технічна документація необхідна для виробництва) і плати копіює і творчо переробляє велика кількість виробників. Обмеження стосується тільки назви «Arduino» – його не можна використовувати для назви не італійських плат.

Все, що розроблено не компанією Arduino, можна умовно розділити на три групи: «клони» , «сумісні» і «Ардуіно-подібні» .

**2.2.2 Клони**

Клонами вважаються плати Arduino, що виготовляються сторонніми компаніями з використанням офіційної документації. Іноді змінюється колір та назва готового продукту. Такі плати (клони) повністю повторюють Arduino і повністю сумісні з нею. Тобто, різниця між клоном і оригіналом тільки в виробнику – відповідно відмінності можуть бути тільки в якості зборки. В цілому, забезпечити гідну якість може китайський виробник, тому покупка оригінальних плат, зазвичай, несе за собою лише зайві витрати.

**2.2.3 Сумісні**

Деякі виробники пішли не шляхом копіювання, а вирішили щось додати в проект і розробили велику кількість своїх плат повністю сумісних з Ардуіно – умовно їх можна назвати «сумісними переробками» (переосмисленнями). Наприклад: Freeduino, Freetronics, Eleven, Seeeduino, CraftDuino, Diavolino, Japanino і багато інших.

Як правило доопрацювання і переробки носять досить естетичний характер (не несуть принципових змін функціоналу або характеристик) інакше плати втратили б сумісність. Зазвичай це додаткові роз'єми, інше розташування світлодіодів і кнопок, своя розводка, застосування інших компонентів (в інших корпусах, інших розмірів), інші схеми живлення, відмінний USB-роз’єм. Є версії міні- нано- та мега- сумісних переробок.

Сумісність з Ардуіно складається з двох речей:

1.Сумісність з платами розширеннями - Шілд. Для цього розташування і вид роз'ємів повинен бути як на оригінальній Arduino.

2.Програмна сумісність. (програмна частина проекту Ардуіно – це середовище розробки (IDE), бібліотеки та скетчі)

На платах Ардуіно встановлені мікроконтролери фірми Atmel, сімейства ATMega - ATMega8 / 168 / 328 – на всіх окрім Мега (ATMega1280 / 2560) і Леонардо (ATMega32U4).

Зазвичай тактування МК відбувається кварцовим резонатором на 16МГц (Рідше 8МГц )

Живлення МК на платах відбувається від джерела 5В (рідше 3,3В)

Завантаження скетчів відбувається через бутлоадер (спеціальна програма-завантажувач заздалегідь прошита в МК), хоча в останніх версіях середовища з'явилася можливість прошивки скетчу через програматор.

Тобто, будь-яка плата, що задовольняє перерахованим умовам (Тип контролера, частота, напруга живлення, наявність бутлоадера) зможе використовувати всі напрацювання спільноти Arduino – скетчі, бібліотеки, і писати все можна буде в тому ж середовищі Arduino, і завантажувати звідти ж.

При деяких допрацюваннях, можна модернізувати бібліотеки для використання не в середовищі Ардуіно або середовище для використання плат з нехарактерними МК або частотами їх роботи. Надалі будемо вважати програмно сумісними тільки ті плати, які коректно запрацюють без всяких допрацювань.

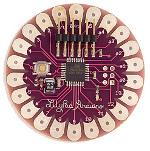
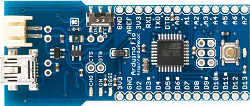
**2.2.4 Ардуіно-подібні**

Деякі виробники вносять більш суттєві зміни втрачаючи сумісність з Шілд, такі плати можна назвати Ардуіно-подібними. Розглянемо найбільш розповсюджені з них:

ArdinoFio (рис. 2.3 а) – плата для портативних пристроїв з живленням від літієвих батарей.

[LilyPad](https://translate.google.com/translate?hl=uk&prev=_t&sl=ru&tl=uk&u=http://arduino.cc/en/Main/ArduinoBoardLilyPad) (рис. 2.3 б) – кругла плата для «електронної одягу»

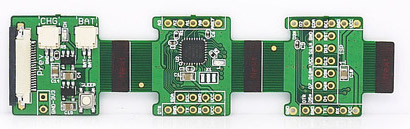
[ProMini](https://translate.google.com/translate?hl=uk&prev=_t&sl=ru&tl=uk&u=http://arduino.cc/en/Main/ArduinoBoardProMini) (рис. 2.3 в) – значною мірою перероблена версія ArdinoMini.



а б в

Рисунок 2.3. – Зовнішній вигляд Ardino-подібних плат ArdinoFio (а), [LilyPad](https://translate.google.com/translate?hl=uk&prev=_t&sl=ru&tl=uk&u=http://arduino.cc/en/Main/ArduinoBoardLilyPad) (б), [ProMini](https://translate.google.com/translate?hl=uk&prev=_t&sl=ru&tl=uk&u=http://arduino.cc/en/Main/ArduinoBoardProMini) (в)

Менш відомі приклади – Roboduino (рис. 2.4 а) – плата для управління великою кількістю сервоприводів. Незважаючи на схожий з Ардуіно вигляд – Шілд з нею не сумісні. Менш схожа на родоначальника [Rainbowduino](http://www.seeedstudio.com/wiki/Rainbowduino_v3.0) – для управління світлодіодними матрицями, Ні на що не схожа версія Seeeduino Film (рис. 2.4 б) від [Seeedstudio](https://translate.google.com/translate?hl=uk&prev=_t&sl=ru&tl=uk&u=http://www.seeedstudio.com/depot/). Є ще велика кількість різноманітних плат подібних Ardino, кожна із них має свої особливості і спеціально заточена під використання із певними апаратними засобами.

а б

Рисунок 2.4. – Зовнішній вигляд Ardino-подібних плат Roboduino (а) і Seeeduino Film (б)

**2.3 Підготовка до роботи з Arduino**

Перш ніж почати експериментувати і програмувати, потрібно виконати деяку попередню підготовку. На персональному комп'ютері потрібно установити драйвер для віртуального СОМ-порту (послідовний інтерфейс) і середовище програмування / розробки для Arduino.

**2.3.1 Установка драйвера**

Спочатку потрібно встановити драйвер для мікросхеми перетворювача USB–UART. Якщо ви використовуєте оригінальну плату Arduino, або високоякісний клон – необхідно встановити драйвер мікросхеми FT232RL, хоча дану операцію можна і не проводити, оскільки цей драйвер вже входить до комплекту інсталятора середовища розробки Arduino. У випадку використання сумісних плат необхідно встановити драйвер перетворювача мікросхеми, що встановлена в ній (здебільшого використовуються мікросхеми CH340, CP2102 або PL2303). Мікроконтролер перетворювача інтерфейсів запрограмований таким чином, що в менеджері пристроїв плата позначена як віртуальний СОМ-порт (віртуальний послідовний інтерфейс). При підключенні плати комп'ютер отримує додатковий СОМ-інтерфейс. ОС Windows не робить різниці між фізичним та віртуальним СОМ-інтерфейсом. Новий інтерфейс отримує наступне вільне ім'я, наприклад, СОМ7.

**2.3.2 Установка програмного забезпечення Arduino**

Тепер встановимо середовище програмування Arduino, яке базується на програмі Processing. Processing – це просте інтегроване середовище розробки, розроблене спеціально для користувача, який не глибоко володіє програмуванням, але, тим не менше, потребує написання власної програми. Останню стабільну версію середовища можна завантажити з офіційного сайту проекту <https://www.arduino.cc/> у вигляді встановлювача, або ж архіву з розпакованою версією, яка не потребує встановлення (портативна версія). Важливо регулярно виконувати оновлення програмного забезпечення, оскільки в середовище постійно додаються нові бібліотеки, драйвери а також виправляються наявні помилки.

Інтегрована середовище розробки (IDE – Itegrated Development Environment) Arduino – це кроссплатформенний додаток на Java, що включає в себе редактор коду, компілятор і модуль передачі прошивки в плату.

Середовище розробки спроектоване для програмування новачками, які не знайомі близько з розробкою програмного забезпечення. Мова програмування аналогічна використовуваному в проекті Wiring. Строго кажучи, це C/C ++ , доповнений деякими бібліотеками та з певними обмеженнями (наприклад недоступні можливості багатопоточного програмування та відсутня об’єктна модель С++). Програми обробляються з допомогою препроцесора, а потім компілюється за допомогою вільно розповсюджуваного компілятора AVR-GCC.

**2.3.3 Середовище розробки Arduino**

Після встановлення або копіювання з архіву програми Arduino можна запустити файл Arduino.exe. Для зручності виклику програми можна створити ярлик на робочому столі. В Arduino-IDE знаходяться різні інструменти і настройки , які полегшують спілкування з програмою Arduino (рис. 2.5).

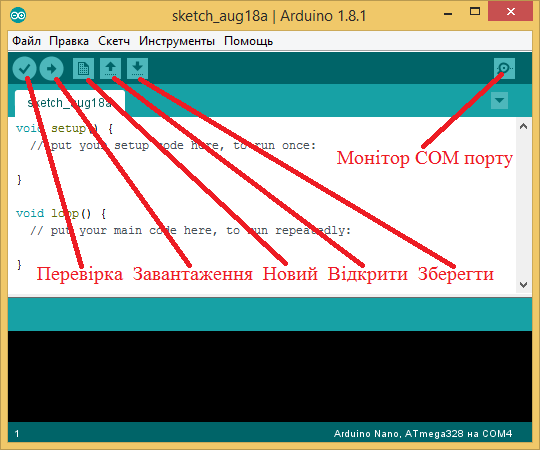


Рисунок. – 2.5. Середовище розробки Arduino

Розглянемо докладніше середовище розробки Arduino. З меню можна викликати всі функції Arduino, далі ми ще будемо знайомитися з окремими його пунктами. Під головним меню знаходиться панель інструментів, де розташовані такі команди:

* *Перевірка* – виконується перевірка на наявність помилок в програмному коді;
* *Завантаження* – формування файлу, який буде переданий в плату мікроконтролера та його завантаження.
* *Новий* – створення нового файлу Arduino.
* *Відкрити* – відкриття тексту програми .
* *Зберегти* – збереження тексту програми .
* *Монітор COM порту* – відкриття вбудованого командного терміналу.

До початку роботи необхідно задати вихідні установки. Для цього потрібно вибрати потрібну плату Arduino і використовуваний інтерфейс (рис. 2.6). У даному випадку вказана плата Arduino Nano. Якщо потрібна інша, то слід вибрати відповідну плату зі списку.

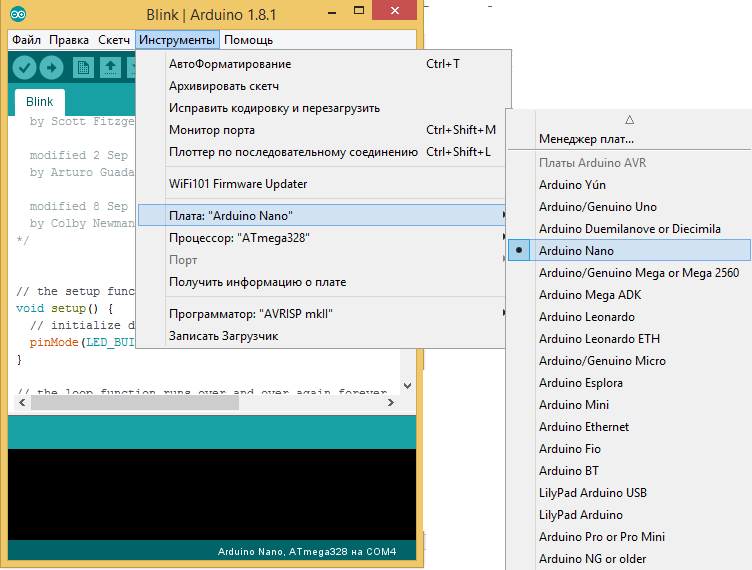


Рисунок 2.6. – Вибір плати мікроконтролера

**ТЕМА 3. ОСНОВИ ПРОГРАМУВАННЯ ARDUINO**

**3.1 Біти і байти**

На даний момент найменша одиниця вимірювання інформації в обчислювальній техніці – це біт. Англійською мовою двійкові числа позначаються як binary digit (двійкова цифра). Так виникло штучно утворене слово "біт" (Bit). Один біт може приймати тільки два значення: 0 або 1 (наявність або відсутність електричного струму). Інформація може представлятися як послідовність з певного числа біт. У нашому випадку 1 байт – це послідовність з 8 біт. Отже, один байт може представлятися і інтерпретуватися як 256 різних комбінацій. Половину байта, тобто 4 біта, називають тетрадою або напівбайтом:

4 біт = 1 напівбайт = 1 тетрада;

8 біт = 2 напівбайта = 1 байт.

Коефіцієнт перерахунку між двійковими одиницями виміру кількості інформації дорівнює 1024 (табл. 3.1). Виняток – перетворення біта в байт, так як при цьому коефіцієнт перерахунку дорівнює 8. Ці некруглі числа виникають через те, що в інформатиці обчислення проводяться за основою "2" (2n ).

*Таблиця 3.1.*

**Співвідношення між одиницями вимірювання кількості інформації**

|  |  |
| --- | --- |
| Найменування | Коефіцієнт перерахунку |
| 1 байт | 8 біт |
| 1 кілобайт (Кбайт) | 1024 байт (102 байт) |
| 1мегабайт (Мбайт) | 1024 Кбайт (102 Кбайт) |
| 1гігабайт (Гбайт) | 1024 Мбайт (102 Мбайт) |
| 1терабайт (Тбайт) | 1024 Гбайт (102 Гбайт) |

**3.2 Базова структура програми**

Структура більшості програм дуже схожа, оскільки широко використовуєся процедурне програмування. Процедурне програмування полягає в розбитті комп'ютерних програм на невеликі приватні завдання, які позначаються як процедури (підпрограми). Найменший і неподільний крок при цьому методі – це команда (оператор, інструкція). Програма виконується послідовно від команди до команди. Сам термін "процедура" означає "просуватись" і походить від латинського слова "procedo". Програміст наказує мікроконтролеру за допомогою програми, в якій послідовності потрібно діяти. Мета цього принципу програмування – простота розробки і можливість повторного використання фрагментів (модулів) вихідного коду.

**3.3 Послідовне** **виконання програми**

При послідовному програмуванні код, що складається з окремих процедур, виконується в циклі. На рис. 3.1 показано виконання послідовності команд "введення даних-обробка-виведення даних".

**3.4 Переривання виконання програми**

Іноді виникає необхідність перервати послідовне виконання програми. Це виконується за допомогою переривань (lnterrupt). Основна частина програми виконується, як і при послідовному програмуванні – нескінчений цикл, в якому повторюються окремі процедури. Як тільки відбудеться зовнішнє або внутрішнє переривання, наприклад, натискання кнопки, основний цикл (Main Loop) перерветься і відбудеться перехід в програму обробки переривань (lnterrupt-Routine). У ній відпрацьовуються

Рисунок 3.1 – Послідовне виконання програми

Початкові дані

Обробка даних

Видача результатів

Цикл

Активація переривання

Допоміжні операції

Цикл

Обробка даних

Видача результатів

Рисунок 3.2 – Переривання виконання програми

спеціальні завдання, наприклад аварійне відключення і т. п. Далі робота триває знову в основному циклі, де можуть виконуватися допоміжні операції. Виконання переривань ілюструє рис. 3.2.

**3.5 Структура програми Arduino**

Програма Arduino складається з наступних основних частин:

* Коментарі та опис програми.
* Заголовки файлів і підключення бібліотеки.
* Оголошення глобальних змінних.
* Стандартна настройка void setup () (порти і конфігурація).
* Основний цикл void loop ().
* Власні процедури.

Як ми побачимо далі, дотримуватися деяких основних правил програмування зовсім не важко. Наступний крок – застосування вже вивченого матеріалу для написання невеликої програми.

Тепер створимо першу справжню програму Arduino. В якості вправи можна просто передрукувати вихідний текст лістингу 3.1.

Лістинг 3.1 – Перша програма для контролера Arduino

// Перша програма Arduino :-)

int ledPin = 13; //Світлодіод під’єднано до цифрового виводу 13 (створюємо глобальну змінну ledPin і присвоюємо їй значення «13»)

void setup() //блок настройки

{

Serial.begin(9600); //ініціалізація послідовної передачі даних на швидкості 9600 бод

pinMode(ledPin, OUTPUT); //встановлення цифрового порту ledPin як «вихід»

Serial.println("First Arduino Program"); //відсилаємо повідомлення на віртуальний COM порт

Serial.println();

}

void loop(){ //Основний цикл

Serial.print("The sum 5 + 188 is ");

Serial.print(5+188);

while(true) //Нескінчений цикл

{

digitalWrite(ledPin, HIGH); // Вмикаємо світлодіод

delay(500); //Чекаємо 500 мс

digitalWrite(ledPin, LOW); //Вимикаємо світлодіод

delay(500); // Чекаємо 500 мс

continue;

}

}

З натисканням кнопки скидання програма запускається знову.

Запустіть термінал після компіляції вихідного коду і передачі в контролер. Наша програма розраховує суму двох чисел і виводить результат через термінал. Після виведення результату починає блимати світлодіод.

Програма ілюструє найважливіші принципи послідовного алгоритму. Спочатку контролеру Arduino повідомляється, що змінна ledPin повинна містити число 13. Потім виконується установка початкових значень. Тут вказується, до якого виводу приєднується світлодіод і задається швидкість послідовного інтерфейсу, рівна 9600 бод. Установка завершується виведенням інформації через послідовний інтерфейс. Далі йде перехід до основного циклу з ім'ям loop(). У ньому перша функція Serial.print виводить текст, а друга - суму 5+ 188. Пам'ятайте, що функція Serial.println після виведення здійснює перехід до лівого краю з передачею управління на початок наступного рядки (CR + LF), а Serial.print – ні.

Після виведення інформації запускається нескінченний цикл while, і виконується безперервне миготіння світлодіода.

**3.6 Команди Arduino і їх застосування**

**Коментарі в початковому тексті**

Щоб правильно читати програму по закінченні певного часу і розуміти її, необхідно ретельно і акуратно документувати свій вихідний код. Можна впроваджувати документацію прямо в вихідний код. Для цього є різні символи коментарів, які призначені для включення звичайних текстів (лістинг 3.2). У середовищі розробки Arduino текст коментарів виділений сірим кольором.

Лістинг 3.2 – застосування коментарів у вихідному коді

//Так виглядає одностроковий коментар

/\*

А так пишуться

Багатострокові коментарі

\*/

Коментарі в коді програми значно полегшують її читання через деякий час. Подумайте, чи зможете ви пригадати через рік, що робили сьогодні в програмі?

**Фігурні дужки** {}

Фігурні дужки виділяють фрагмент коду для компілятора.

Блок коду завжди відкривається символом { і завершується }. Між дужками розташовуються команди.

type Main\_Function ()

{

Між дужками знаходяться команди

}

**Крапка з комою ;**

Крапка з комою завершує команду. Якщо пропустити цей символ, компілятор Arduino видасть повідомлення про помилку.

int х = 42; // Тут змінна х оголошується як Intеgеr і їй присвоюється число 42, крапка з комою завершує визначення.

**Типи даних і змінні**

Кожна програма складається з різних змінних, які або залежать від зовнішнього оточення (наприклад, аналоговий або цифровий вхід), або потрібні для розрахунку в програмі і виведення результатів. У розпорядженні програміста є різні типи змінних: Byte (байт), Integer (ціле число), Long (довге ціле число) і Float (число з плаваючою комою). Тип змінної завжди потрібно ставити перед її застосуванням.

**Ім'я змінної**

У середовищі розробки Arduino є відмінність між верхнім і нижнім регістром символів імені змінної. В Arduino допускається підкреслення \_ , воно часто використовується, щоб зробити довгі імена змінної розбірливими. Ключові слова (if, while, do і т. д .) не можуть бути іменами змінної . Імена глобальних змінних і функцій не можуть збігатися. Крім того, не допускається одночасне задання функцій і локальних змінних з одним тим же ім'ям.

**Локальні і глобальні змінні**

Якщо змінна оголошується в межах функції, процедури або як аргумент функції, вона є локальною. Це означає, що змінна існує тільки всередині своєї функції. Змінна, оголошена поза функцією, є глобальною і визначена для всіх функцій в межах нашої програми.

byte Variable; // Змінна типу Byte, вона може набувати значень від 0 до 255

float PI = 3.1415; // Константа PI оголошена як Float (з плаваючою точкою)

int myArray[9]; //Оголошено масив розмірністю 10 елементів типу Integer

**3.7 Типи даних**

Розглянемо які типи даних існують і скільки при цьому займають пам'яті.

***Boolean***

Змінна типу Boolean може приймати два стани: true (істина) або false (брехня). Така змінна займає 1 байт в пам'яті. Значення true відповідає логічній одиниці, а false – це не логічний нуль, як часто думають, а стан, відмінний від логічної одиниці.

boolean bVar = true; // Змінна істинна

***Byte***

1 байт – це 8 біт, тому така змінна може приймати значення від 0 до 255.

byte myVariable = 0; // Змінна ініціалізується нульовим значенням

***Char***

Символ має розмір l байт. Змінна Char – це символ в одиночних кавичках (апострофах). Якщо потрібно рядок символів, то їх розміщують в подвійних лапках (наприклад, "ПРИВІТ"). Символ отримує номер з набору символів ASCII. Наприклад, буква "А" - це число 65. Символьні змінні можуть набувати значень від-127 до +127.

char myChar = "А"; // Число 65

***Unsigned Char***

Символи без знака (unsigned Char) поводяться як символи зі знаком, але вони можуть приймати тільки позитивні значення в діапазоні від 0 до 255.

unsigned char myChar = "В"; // Число 66

***lnt (lnteger)***

Ціле число (integer) складається з двох байт і може приймати значення від –32768 до +32768.

int myIntegerVariable = -2973; // Ціла змінна зі значенням -2973

***Unsigned int***

Тип unsigned integer охоплює змінні в діапазоні від 0 до 65 535 (216 –1).

навідміну від int тип unsigned int не має знака. Така змінна займає у пам'яті два байта.

unsigned int my unsignedIntVariable = 50000; //Ціла змінна без знака зі значенням 50000

***Long***

Змінна типу Long складається з чотирьох байт і може приймати значення від –2147483648 до 2147483647.

long bigNumber = 10000000; // змінна Long зі значенням 10000000

***Unsigned Long***

Змінна unsigned Long складається з чотирьох байт і може приймати значення від 0 до 4294967295. Змінна не має знака і може зберігати тільки позитивні значення.

unsigned long veryBigNum = 54544454544; // Дуже, дуже велика змінна

***Float***

Змінні Float можуть зберігати 32 біта значення зі знаком. Діапазон можливих значень лежить в межах від –3.4028235Е+38 до +3.4028235Е+38. Для цієї змінної потрібно чотири байти в пам'яті.

float Var = 100.42; // змінна Float зі значенням 100,42

***String***

Строкова змінна (string) - це масив (Array) змінних Char і нульового символу.

Для кожного символу потрібно, таким чином, один байт і в кінці ланцюжка додатково ще один байт, для нульового символу. Таким чином, наприклад, для слова "Hallo" потрібно шість байтів.

char myString [] = "Hallo Max"; // Потрібно 10 байт

***Arrays***

Масив (Arrays) – це упорядкований набір змінних одного типу (рис. 3.3). В інформатиці масив позначає структуру даних. За допомогою масиву дані, однакового типу (byte, int і т.д.) заносяться в пам'ять комп'ютера таким чином, що доступ до даних стає можливий через покажчик. У мові Arduino нумерація елементів масиву починається з нуля, в деяких інших компіляторах нумерація починається з одиниці. Крім того, важливо знати, що Arduino насьогодні підтримує тільки одномірні масиви. Приклад роботи з масивами приведений в лістингу 3.3.

Індекс 0

Індекс 1

Індекс 2

Індекс n

Область пам’яті масиву

Рисунок 3.3 – Структура масиву

Лістинг 4.3 –Приклад роботи з масивами

//Arrays

int Array\_1 [3];

int Array\_2 [] = {1, 2, 3};

void setup () {

Serial.begin (9600);

Serial.println ( "Arduino Arrays");

Serial. println ();

}

void loop () {

byte х;

Array\_l [0] = 1;

Array\_l [l] = 2;

Array\_l [2] = 3;

Array\_l [3] = 4;

Serial.println ( "Array 1");

Serial.println “-----------------------------”);

//Видача даних першого масиву

for (x = 0; x <3; x ++) Х

Serial.print (Array\_l [х]);

Serial.println ();

}

Serial.println ( "Array 2");

Serial.println “-----------------------------”);

//Видача даних другого масиву

Serial.print (Array\_2 [0]);

Serial.println ();

Serial.print (Array\_2 [1]);

Serial.println ();

Serial.print (Array\_2 [2]);

while (l);

У першому масиві ми задаємо розмір "3". Таким чином, в цьому масиві можна зберігати чотири змінні типу Integer (16 біт ). Розмірність другого масиву не вказано і три його елементи задані в фігурних дужках. Мова йде про динамічний масив. В основному циклі ми призначаємо елементам першого масиву різні значення, в нашому випадку 1, 2, 3, 4. Можете вказати інші значення , щоб краще зрозуміти роботу з масивами .

Змінна лічильника х циклу for містить значення покажчика масиву для виведення значень в терміналі. Значення елементів другого масиву і покажчика теж виводяться в терміналі.

Цикл while (1); представляє собою нескінченний цикл основної програми, для її одноразового запуску.

**3.8 Оператори**

Кожному типу даних відповідають свої оператори, які вказують, на операції можуть, що застосовуватися. Розглянемо можливі оператори Arduino і їх дію.

**Арифметичні оператори**

= (Присвоювання).

+ (Додавання).

– (Віднімання).

\* (Множення).

/ ( Ділення ).

% ( Ділення по модулю – обчислення залишку від ділення ).

**Оператори порівняння**

== (Дорівнює, наприклад А == в).

!= (Не дорівнює , наприклад А != в ).

< (Менше , наприклад А < в ).

> (Більше , наприклад А > в ).

<= (Менше дорівнює , наприклад А <= в ).

> = (Більше дорівнює , наприклад А > = в ).

**Побітова арифметика**

& (Бітове І).

| (Бітове АБО ).

~ (Бітове НІ ).

**Булева арифметика**

&& (Логічне І , наприклад , якщо вираз && Unswer\_b істиний, тоді виконувати якусь дію).

|| (Логічне АБО, наприклад, якщо вираз || Unswer\_b істиний, тоді виконувати якусь дію).

! (Логічне заперечення, наприклад, якщо вираз !Unswer\_b істиний, тоді виконувати якусь дію).

**Збільшення і зменшення значення**

++ (Інкремент, наприклад i++, змінна i буде збільшена на одиницю).

–– (Декремент, наприклад i--, змінна i буде на одиницю менше).

+= (Присвоювання з інкрементом, наприклад i+=5, змінна i буде збільшений на 5 і результат присвоєно змінній i).

–= (Присвоювання з декрементом, наприклад i–=5, змінна i буде змен шена на 5 і результат присвоєно змінній i).

\*= (Присвоювання з множенням, наприклад i\*=2, змінна i буде помножена на на 2 і результат присвоєно змінній i).

/*=*(Присвоювання з діленням, наприклад: i/=2, змінна i буде розділена на 2 і результат присвоєно змінній i).

**Константи**

Приклади констант:

НIGH / LOW (НIGH = 1, LOW = О).

INPUT / OUTPUT (INPUT = О, OUTPUT = 1).

true / false (true = 1, false! = 1).

**Цілочисельні константи**

Цілочисельні константи в коді – це числа, наприклад, 123. За замовчуванням ці числа інтерпретуються як цілі типу int, однак ви можете змінити це за допомогою модифікаторів U і L.

Щоб задати цілочисельний константі інший тип, потрібно після неї записати модифікатор:

* 'u' або 'U ", щоб привести константу до беззнакового типу даних. Наприклад: 33u
* 'l' або 'L', щоб привести константу до типу даних long. Наприклад: 100000L
* 'ul' або 'UL', щоб привести константу до типу unsigned long. Наприклад: 32767ul

Цілочисельні константи інтерпретуються як числа в десятковій системі числення, тому для задання числа в іншій системі необхідно використовувати спеціальні префікси (табл. 3.2).

*Таблиця 3.2*

**Приведення констант до різних систем числення**

|  |  |  |  |
| --- | --- | --- | --- |
| Основа системи | Приклад | Префікс | Коментар |
| 10 (десяткова) | 618 | немає |  |
| 2 (двійкова) | B1111011 | "B" | працює тільки з 8-бітними значеннями (0 - 255). Коректні символи 0-1 |
| 8 (восьмерична) | 0173 | "0" | коректні символи 0-7 |
| 16 (Шістнадцятерична) | 0x7B | "0x" | коректні символи 0-9, A-F, a-f |

Основа десяткової системи числення – 10. Математичні операції з такими числами всім знайомі. Константи без будь-яких префіксів вважаються десятковими.

Приклад:

101 // те ж саме, що і 101 в десятковій системі ((1 \* 10 ^ 2) + (0 \* 10 ^ 1) + 1)

Основа двійковій системи числення – 2. Для запису чисел в цій системі використовуються тільки 0 і 1.

Приклад:

B101 //те ж саме, що і 5 в десятковій системі ((1 \* 2 ^ 2) + (0 \* 2 ^ 1) + 1)

Префікс двійковій системи може використовуватися тільки з числами розмірністю 1 байт (8 біт) в діапазоні від 0 (B0) до 255 (B11111111). Для запису числа типу int (16 біт) в двійковому форматі можна використовувати подвійну операцію:

myInt = (B11001100 \* 256) + B10101010; // B11001100 - старший байт.

Основа восьмиричної системи числення – 8. Для запису чисел в цій системі використовуються тільки цифри в діапазоні від 0 до 7. Вісімкові числа характеризуються префіксом "0".

Приклад:

0101 // те ж саме, що і 65 в десятковій системі ((1 \* 8 ^ 2) + (0 \* 8 ^ 1) + 1)

Увага. Можна допустити помилку, що дуже складно знаходиться, якщо ненавмисно дописати 0 перед константою, оскільки в цьому випадку остання буде інтерпретуватися компілятором як вісімкове число.

Основа шістнадцятирічної системи числення – 16. Для запису чисел в цій системі використовуються цифри від 0 до 9, а також літери від A до F; при цьому A має значення 10, B - 11, і т.д. до F, яке еквівалентно 15. Шістнадцяткові значення характеризуються префіксом "0x". Зверніть увагу, що A-F можуть вводиться як у верхньому, так і нижньому регістрах (a-f).

Приклад:

0x101 // те ж саме, що і 257 в десятковій системі ((1 \* 16 ^ 2) + (0 \* 16 ^ 1) + 1)

**Константи з плаваючою точкою**

Подібно цілочисельним констант, константи з плаваючою точкою використовуються для того, щоб зробити програмний код більш читабельним. При компіляції коду дробові константи замінюються їх числовими значеннями. Наприклад n = .005;

Константи з плаваючою точкою також можуть бути виражені в експоненційної формі. Обидва символи 'E' і 'e' інтерпретуються як показники експоненти.

Приклад:

2.34E5 = 2.34 \* 10 ^ 5 = 234000;

67e-12 = 67.0 \* 10 ^ -12 = .000000000067

**Директива #define**

Define – це директива препроцесора, яка запускається перед компіляцією. Можна сказати, що мова йде про власний маленький компілятор, який заздалегідь перетворює команди #define в константи. Директива дозволяє задавати постійне значення змінної:

#define Variable 1 // Без крапки з комою

Тут змінна Variable отримує значення 1. Кожен раз, коли в програмі зустрічається ім'я Variable, значення змінної замінюється на 1. Для нас ця заміна неочевидна, але компілятор діє так, як описано.

**3.9 Керуючі конструкції**

Для реагування на події (умови) у кожній програмі використовуються, так звані, керуючі конструкції. Вони мають конструкцію, аналогічну тим, що використовуються в мовах С і С++ і мають вигляд if... else-if... else або switch case.

В наведених далі лістингах виведення даних здійснюється через послідовний порт Arduino і мужуть бути зчитані через термінальну програму (монітор COM-порту, Putty Terminal, або будь-яку іншу). Завантажте приклади в плату, мікроконтролера і запустіть після цього термінальну програму, щоб розібратися, що відбувається.

**Оператор if**

Лістинг 3.4 ілюструє синтаксис, а лістинг 3.5 – приклад використання оператора if.

Лістинг 3.4 – Синтаксис конструкції if

if (VarА == VarВ) {

//Тут знаходиться код, який виконується коли VarА дорівнює VarВ

}

Лістинг 3.5 – Приклад використання оператора if

// конструкція if

int x;

void setup(){

Serial.begin(9600);

Serial.println("If Construction");

Serial.println();

}

void loop(){

if(x==10) {

Serial.println("Змінна «х» стала рівною 10");

while(1);

}

x++;

}

Основний цикл void loop () виконується до тих пір, поки змінна х типу Integer не дорівнюватиме 10. Тіло циклу складають оператори між if і дужками {}. За допомогою if можна реалізувати простий спосіб передачі управління (розгалуження в програмі). Ви можете самі експериментувати з іншими операторами. Тут, також, допустимі логічні операції !=, <, > і т. п.

***if*** ... ***else***

Лістинг 3.6 пояснює синтаксис, а лістинг 3.7 – приклад використання оператора if ... else.

Лістинг 3.6 – Синтаксис конструкції if … else

if (VarА> VarВ){

//Код, який повинен виконуватися коли VarА більше VarВ

}

else {

// Код, який повинен виконуватися коли VarА не більше VarВ

}

Лістинг 3.7 – Приклад використання оператора if … else

//Конструкція if... else...

int x;

void setup() {

Serial.begin(9600);

Serial.println("If and Else ");

Serial.println();

}

void loop() {

if(x==10){

Serial.println("Змінна «х» стала рівною 10!");

while(1);

}

else{

Serial.print("X = ");

Serial.println(x);

}

x++;

}

Конструкцією else можна реалізувати різні альтернативні варіанти. Програма виводить значення змінної x. поки воно не досягне 10. Як тільки змінна х буде дорівнювати 10, запускається фрагмент програми між дужками

***else if {}.***

Можливість багаторазового чергування команд дає оператор else if. Тут можуть запитуватися різні стани змінних. Залежно від логічного значення (true або false) в умовному операторі else if виконується відповідна гілка програми. Синтаксис пояснює лістинг 3.8.

Лістинг 3.8 – Синтаксис конструкції else if

if (VarА != VarВ) {

//Код, що виконується

}

else if (VarА == VarВ) {

//Код, що виконується

}

else (VarА > VarВ) {

//Код, що виконується

}

Лістинг 3.9 в залежності від значення змінної х виводиться різний текст.

Лістинг 3.9 – Приклад позгалудження програм за допомогою конструкції else.. if

//Конструкція else if..

int x;

void setup() {

Serial.begin(9600);

Serial.println("Else If ");

Serial.println();

}

void loop() {

if(x==65){

Serial.println("Змінна «х» стала рівною 65!");

while(1);

}

else if(x==10){

Serial.println("Змінна «х» стала рівною 10!");

while(1);

}

else{

Serial.print("X = ");

Serial.println(x);

}

x++;

}

***switch case***

Конструкція switch case поводиться подібно оператору else if. Тут також залежно від логічного значення відповідного виразу буде виконано відповідний розділ коду. За замовчуванням можна задати альтернативний варіант, при відсутності відповідності умов всередині оператора case. Вихід з оператора case здійснюється за допомогою оператора break (лістинг 3.10). Приклад приведений в лістингу 3.11.

Лістинг 3.10 – Синтаксис конструкції switch case

Switch (Variable) {

case 1:

//Код, який повинен виконатись

case 1:

break;

//Код, який повинен виконатись

break;

default:

//Альтернативний код, який виконуються коли всі інші умови не виконуються

}

Лістинг 3.11 – Приклад використання конструкції switch case

//Switch Case

int x;

void setup() {

Serial.begin(9600);

Serial.println("Конструкція Switch Case");

Serial.println();

}

void loop(){

switch(x)

{

case 10:

Serial.println("значення «х» досягло 10");

break;

case 20:

Serial.println("значення «х» досягло 20");

break;

case 30:

Serial.println("значення «х» досягло 30");

while(1);

break;

default:

Serial.print("X = ");

Serial.println(x);

}

x++;

}

В основному циклі значення змінної х збільшується на одиницю. При певній умові в команді switch, в термінал буде виведено повідомлення.

**3.10 Цикли**

При програмуванні часто потрібні цикли, наприклад, щоб реалізувати десятковий, двійковий лічильники або змусити програму нескінченне число разів повторювати виконання деякого фрагмента. Цикл підходить також для організації читання даних з послідовного інтерфейсу. Існують кілька типів циклів. У кожного з них є свої особливості, з якими ми зараз познайомимося.

***for***

Цикл for рахує змінну вгору або вниз в межах одного зазначеного діапазону значень. При цьому може здаватися певна величина наростання (інкремент). Лістинг 3.12 ілюструє синтаксис циклу for, а лістинг 3.13 – приклад його використання.

Лістинг 3.12 – Синтаксис циклу for

for (Умова початку; Умова закінчення; Лічильник) {

//блок програми

}

Лістинг 3.13 – Приклад використання циклу for

// Цикл For

int x;

void setup(){

Serial.begin(9600);

Serial.println("For Construction");

Serial.println();

}

void loop(){

Serial.println("Збільшення на 1");

for(x=0;x<11;x++){

Serial.print("X = ");

Serial.println(x);

}

Serial.println("Збільшення на 2");

for(x=0;x<11;x=x+2) {

Serial.print("X = ");

Serial.println(x);

}

Serial.println("Зменшення змінної від 10 до 1 із кроком 1");

for(x=10;x!=0;x--) {

Serial.print("X = ");

Serial.println (x);

}

// Закінчення програми

while(1);

}

Програма містить три цикли for і пояснює принцип роботи за допомогою виведення показань лічильника. Щоб рахувати до 10, потрібно вказувати максимальне значення лічильника циклу, рівним 11. Таким чином, цикл працює, поки значення лічильника менше 11. Якщо задати максимальне значення рівним 10, то цикл не відпрацює останній крок.

***while* і *do while***

Ще два варіанти циклу – це while і do while. Оператори в тілі циклу do while виконуються один раз, потім перевіряється умова закінчення циклу. Тобто, цей цикл виконається, щонайменше, один раз. Оператор while часто застосовується для організації нескінченних циклів. Проте, цикл можна прервати за допомогою оператора break. Якщо потрібно перевірити умову перед ви виконанням операторів циклу, то потрібно використовувати оператор while. Спочатку проводиться перевірка, а потім виконуються оператори тіла циклу. Лістинг 3.14 ілюструє синтаксис циклу while, а лістинг 3.15 – приклад його використання.

Лістинг 3.14 – Синтаксис циклів while і do while

//Нескінченні цикли

while (l) {

//Працювати за будь-яких умов

}

//Нескінченні цикли з умовою скасування

while (l) {

Var ++;

if (Var> 10) break;

}

// While

While (Var <10) {

Var ++;

}

//Do While

do {

Var ++;

} while (Var <10);

Лістинг 4.15 – Приклад використання циклів while і do while

// While & Do While

int X=0;

void setup(){

Serial.begin(9600);

Serial.println("Конструкції While та Do While ");

Serial.println();

}

void loop() {

while(1) {

X++;

Serial.println (X);

if(X>9) break;

}

X=0;

Serial.println();

while(X<10) {

X++;

Serial.println(X);

}

X=0;

Serial.println();

// Do While

do {

X++;

Serial.println(X);

}while( X < 10 );

while(1); //Кінець програми

}

**3.11Функції та підпрограми**

В процесі програмування дуже часто потрібні функції. Вони роблять програму наочнішою, а також дозволяють реалізувати власні команди. Ви можете створити свої функції і процедури (підпрограми, які не повертають значень), які будуть потрібні знову і знову, і використовувати їх в майбутніх проектах (принцип модульної побудови). Різниця між функцією і процедурою (універсальна назва Sub Routine (підпрограма)) полягає в тому, що функція повертає значення, а процедура ні. У функції може, наприклад, виконуватися математичний вираз, який повертає результат переданої змінної.

**Підпрограма**

Лістинг 3.16 ілюструє приклад підпрограми.

Лістинг 3.16 – Реалізація підпрограм

// Arduino Sub-Routinen

void setup(){

Serial.begin(9600);

Serial.println("Arduino Sub-Routinen");

Serial.println();

}

void loop(){

Issuance1();

Issuance2();

while(1);

}

void Issuance1(){

Serial.println("Issuance 1");

}

void Issuance2(){

Serial.println("Issuance 2");

}

**Функції**

Функція – це блок кодів програми, у яких є ім'я і ряд команд, які запускаються при виконанні функції. Приклади функцій void setup {} і void loop {} вже зустрічалися раніше. Існують також вбудовані функції, які будуть розглянуті далі.

Писати власні функції має сенс, щоб спростити повторювані задачі і покращити структуру програми. При оголошенні вказують тип функції, він ідентичний типу даних, наприклад, int для типів integer. Якщо функція не повертає значення, її тип буде void. Крім типу при оголошенні функції задають ім'я і в дужках перераховують всі параметри, які повинні їй передаватися. Лістинг 3.17 ілюструє приклад функції.

Лістинг 3.17 – Приклад реалізації функції

// Arduino Function

void setup(){

Serial.begin(9600);

Serial.println("Arduino Function");

Serial.println();

}

void loop(){

int Sum;

Sum = Addition(12,55);

Serial.print("12 + 55 = ");

Serial.println(Sum);

while(1);

}

int Addition(int x, int y) {

int sum;

sum=x+y;

return sum;

}

***continue***

За командою continue решта поточного циклу (do, for або while) пропускається і запускається код після блоку {} . Лістинг 3.18 ілюструє приклад.

Лістинг 3.18 – Приклад використання команди Continue

// Arduino Continue

int i=0;

void setup() {

Serial.begin(9600);

Serial.println("Arduino Continue");

Serial.println();

}

void loop(){

for(i=0;i<10;i++) {

if(i%2==0) {

continue;

}

Serial.print(i);

Serial.println(" Не ділиться на 2");

}

while(1);

}

Тут оператор continue завжди перериває цикл for, коли змінна i не де ділиться на 2 без залишку.

**Функції перетворення типу**

Функції char (), byte (), int (), long () і float () перетворять тип змінної. Таким чином, ми можемо, наприклад, з однобайтової змінної зробити змінну типу Long. Суть подібних операцій – приведення типу даних для по наступних обчислень.

Ось перелік функцій перетворення типу:

* Char () – перетворює значення в символ (Charakter).
* Byte () – перетворює значення в Byte.
* Int () – перетворює значення в integer.
* Long () – перетворює значення в Long.
* Float () – перетворює значення в Float.

**Математичні функції**

***min (x, у)***

Розраховує мінімум з двох значень і повертає менше значення. Лістинг 3.19 ілюструє приклад використання цієї функції

Лістинг 3.19 – Приклад використання функції min(x,y)

// Arduino min(x,y) Funсtion

int x,y,Erg=0;

void setup(){

Serial.begin(9600);

Serial.println("Arduino min(x,y) Funсtion");

Serial.println();

}

void loop(){

Erg=min(10,55);

Serial.print(Erg);

Serial.println();

while(1);

}

***тах {х, у)***

Розраховує максимум з двох значень і повертає число з більшим значенням. Ліститинг 3.20 ілюструє приклад використання max(x,y).

Лістинг 3.20 – Приклад використання функції max(x,y)

// Arduino max(x,y) Funсtion

int x,y,Erg=0;

void setup(){

Serial.begin(9600);

Serial.println("Arduino max(x,y) Funсtion");

Serial.println();

}

void loop(){

Erg=max(10,55);

Serial.print(Erg);

Serial.println();

while(1);

}

***abs (x)***

Розраховує абсолютну величину числа. Лістинг 3.21 ілюструє приклад використання abs(x).

Лістинг 3.21 – Приклад використання функції abs(x)

// Arduino abs(x) Function

int Erg;

void setup(){

Serial.begin(9600);

Serial.println("Arduino abs(x,y) Function");

Serial.println();

}

void loop(){

Erg=abs(3.1415);

Serial.print(Erg);

while(1);

}

***constrain (x, а, b)***

Функція перевіряє і якщо треба задає нове значення, так щоб воно було в області допустимих значень, заданої параметрами.

Параметри:

* x –значення, що перевіряється (будь-який тип)
* a – нижня межа області допустимих значень, будь-який тип
* b – верхня межа області допустимих значень, будь-який тип

Значення, що повертається:

* x – якщо x входить в область допустимих значень [a..b]
* a – якщо x менше a
* b – якщо x більше b

Лістинг 3.22 ілюструє приклад перевірки входження числа в заданий діапазон.

Лістинг 3.22 – Приклад використання функції constrain(x, a, b)

// Arduino constrain(x, a, b) Function

int x,Erg;

void setup(){

Serial.begin(9600);

Serial.println("Arduino constrain(x, a, b) Function");

}

void loop(){

for(x=0;x<60;x++){

Erg=constrain(x, 10, 50);

Serial.println(Erg);

}

while(1);

}

***тар (х, fromLow, fromHigh, toLow, toHigh)***

Функція map – це корисна функція для перерахунку значень з одного діапазона в інший. Вона ідеально підходить, наприклад, для перетворення великої вхідний змінної в маленьку вихідну змінну. Лістинг 3.23 ілюструє приклад перерахунку з використанням функції map.

Лістинг 3.23 – Приклад використання функції map

// Arduino map(x, fromLow, fromHigh, toLow, toHigh) Function

int x, Erg;

void setup(){

Serial.begin(9600);

Serial.println("Arduino Map Function");

Serial.println();

}

void loop(){

for(x=0;x<20;x++){

Erg=map(x,0,20,5,15);

Serial.println(Erg);

}

while(1);

}

***pow (base, exponent)***

Функція pow видає результат піднесення до степеня першого значення аргументу з другим значенням аргументу (перше значення аргументу – число, друге – степінь). У лістингу 3.24 обидва аргументи і результат мають тип float.

Лістинг 3.24 – Приклад піднесення числа до степеня функцією pow(base,exponent)

// Arduino pow(base,exponent) Function

int Erg;

void setup() {

Serial.begin(9600);

Serial.println("Arduino pow(base,exponent) Function");

Serial.println();

}

void loop(){

Erg=pow(10,5);

Serial.println(Erg);

while(1);

}

***sq (x)***

Функція обчислює квадрат аргументу. Лістинг 3.25 ілюструє приклад роботи функції sq(x).

Лістинг 3.25 – Приклад піднесення числа до степеня 2 функцією sq(x)

// Arduino sq(x) Function

int Erg;

void setup(){

Serial.begin(9600);

Serial.println("Arduino sq(x) Function");

Serial.println();

}

void loop(){

Erg=sq(9);

Serial.print(Erg);

Serial.println();

while(1);

}

***sqrt (x)***

Ця функція, зворотна sq (х), вона обчислює квадратний корінь числа. Лістинг 3.26 ілюструє приклад її роботи.

Лістинг 3.26 – Приклад знаходження квадратного кореня числа функцією sqrt(x)

// Arduino sqrt(x) Function

int Erg;

void setup(){

Serial.begin(9600);

Serial.println("Arduino sqrt(x) Function");

Serial.println();

}

void loop(){

Erg=sqrt(9);

Serial.print(Erg);

Serial.println();

while(1);

}

***sin (rad)***, ***cos (rad), tan (rad)***

Функції розраховують синус, косинус і тангенс числа відповідно. Аргумент вказується в радіанах. Значення, що повертається – це синус, косинус або тангенс аргументу. Лістинг 3.27 ілюструє приклад використання цих функції.

Лістинг 3.27 – Приклад знаходження sin(x), cos(x), tan(x)

// Arduino sin(x), cos(x), tan(x) Function

float ErgS, ErdC, ErgT, x=1.0;

void setup(){

Serial.begin(9600);

Serial.println("Arduino sin(x) Function");

Serial.println();

}

void loop(){

ErgS=sin(x);

ErgC=cos(x);

ErgT=tan(x);

Serial.print(“Sin “);

Serial.print(x);

Serial.print(“ = “);

Serial.println(ErgS);

Serial.print(“Cos “);

Serial.print(x);

Serial.print(“ = “);

Serial.println(ErgC);

Serial.print(“Tan “);

Serial.print(x);

Serial.print(“ = “);

Serial.println(ErgT);

while(1);

}

**РОЗДІЛ 4. ПРИКЛАДНЕ ПРОРАМУВАННЯ В СЕРЕДОВИЩІ ARDUINO**

**4.1 Послідовний інтерфейс введення / виведення**

Зв'язок через інтерфейс UART застосовується досить часто. Мікроконтролер може посилати, а також приймати дані від комп'ютера або інших мікроконтролерів.

В Arduino для цього існує кілька команд. Деякі вже зустрічалися у попередніх прикладах, наприклад, Serial.print () і Serial.println (). Мікроконтролер має вбудований апаратний інтерфейс UARТ. Універсальний асинхроннний прийоомо-передавач (UART - Universal Asynchronous Receiver Transmitter) можна також імітувати за допомогою програмного забезпечення. Емулятор буде не настільки швидкісний, як апаратний інтерфейс, але всетаки дозволяє встановлювати одночасне підключення до декількох станцій. На рис. 4.1 зображений апаратний інтерфейс UART, реалізований в Arduino.

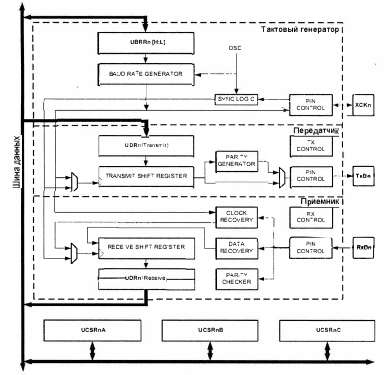


Рисунок. 4.1 – Апаратний інтерфейс UART мікроконтролера

Arduino кoмaндa Serial.println посилає послідовність символів (так званий рядок) через інтерфейс UART. "Невидимі" символи CR (Carriage Return) і LF (LineFeed) додаються автоматично і позначають кінець рядка. Якщо перехід на інший рядок не потрібний, то слід вказати команду Serial.print. Якщо в Serial.print передається число, воно автоматично перетворюється в текст. Передається не число, а АSСII-код для цього числа. Наприклад, число 42 складається з двох символів 4 і 2. Будуть передані два АSСII-коди для символів "4" і "2" і АSСII-коди обох службових символів "CR" і "LF". Важливо, щоб приймачі і передавачі були завжди встановлені на однакову швидкість в бодах.

**4.1.1 Функції роботи з послідовним інтерфейсом**

***Serial.begin (Baudrate)***

Функція Serial.begin() задає швидкість послідовного інтерфейсу в бодах. Бод позначає число біт в секунду. Якщо ми вибираємо 9600 бод, один біт передається за 1/9600 = 0,000104 с = 104 мкс.

Функція Serial.begin (Baudrate) відкриває послідовний порт і встановлює швидкість в бодах (швидкість передачі даних) для послідовної передачі даних. Можливі такі стандартні швидкості в бодах:

* 300
* 1200
* 4800
* 9600
* 14400
* 19200
* 28800
* 38400
* 57600
* 115200

При використанні плати Arduino Mega або іншої з декількома апаратними інтерфейсами UART, конфігурацію можна задати таким чином:

Serial.begin (9600);

Seriall.begin (38400);

Serial2.begin (19200);

Serial3.begin (4800);

Всі інтерфейси мають номер. Перший інтерфейс – UART0. Якщо потрібно вивести символ, то в цьому випадку застосовується функція Serialx.print(), де х відповідає номеру інтерфейсу UART.

Serial.println ("Hallo, this is UART 0");

Seriall.println ("Hallo, this is UART 1");

Serial2.println ("Hallo, this is UART 2");

SerialЗ.println ("Hallo, this is UART З");

При послідовній комунікації цифрові виводи 0 (RX) і 1 (ТХ) не можуть використуватися одночасно.

***Serial.end ()***

Для завершення роботи з послідовним інтерфейсом і використання виводів за іншим призначенням, можна застосувати функцію Serial.end ().

***Serial.read ()***

Функція Serial.read() зчитує один байт з послідовного інтерфейсу:

int х;

x = Serial.read();

***Serial.available ()***

Функція Serial.available() вказує, чи є дані в буфері послідовного порту. Ця функція дозволяє, наприклад, пропустити блок програми, якщо у буфері немає ніяких даних. Ця функція дуже важлива для роботи з послідовним інтерфейсом. У лістингу 4.1 показано, як ця функція працює на практиці.

Лістинг 4.1 – Приклад використання функції Serial.available()

// Arduino Serial.available Function

byte eingabe, ausgabe;

void setup(){

Serial.begin(9600);

Serial.println("Arduino Serial.available Function");

Serial.println();

}

void loop(){

if (Serial.available() > 0){

eingabe=Serial.read();

Serial.print("були отримані такі дані: ");

ausgabe=char(eingabe);

Serial.println(eingabe);

}

}

***Serial.flush ()***

Ця функція видаляє вміст послідовного буфера. Застосовують її, щоб очистити буфер після розподілу послідовних даних. Якщо виникає помилка комунікації і дані пошкоджуються, то вони видаляються з буфера.

***Serial.print ()***

Функція Serial.print() виводить дані з послідовного буфера передачі. Допустимі типи Integer, Byte, Char і Float.

Функція serial .print (х) без вказівки формату виводить десяткове число з буфера UART:

int b = 79;

Serial.print (b);

Виводить число 79 з буфера UART.

Функція serial.print (b, DEC) із зазначенням формату DEC виводить число як АSСII-рядок з буфера UART:

int b = 79;

Serial.print (b, DEC);

Виводить АSСII-рядок "79" з буфера UART.

Функція Serial .print (b, Нех) із зазначенням формату Нех виводить з буфера

UART число як АSСII-рядок в шістнадцятирічному форматі:

int b = 79;

Serial.print (b, Нех);

Виводить АSСП-рядок “4F “ з буфера UART.

Функція Serial.print (b, OCT) із зазначенням формату OCT виводить з буфера

UART число як АSСII-рядок в вісімковому форматі:

int b = 79;

Serial.print (b, ОСТ);

Виводить АSСП-рядок “117” з буфера UART.

Функція Serial.print (b, ВIN) із зазначенням формату Вin виводить з буфера

UART число як ASCII-cтpoкy в довічним форматі:

int b = 79;

Serial.print (b, BIN);

Виводить АSСII-рядок ”1001111"з буфера UART.

Функція serial.print (b, Byte)із зазначенням формату Byteвиводить з буфера

UART число у вигляді окремого байта:

int b = 79;

Serial.print (b, Byte);

Виводить АSСII-символ "О" з буфера UART.

Лістинг 4.2 ілюструє приклади виведення в різних форматах.

Лістинг 4.2 – Виведення чисел у різних форматах

// Arduino Serial.print Function

int x;

void setup(){

Serial.begin(9600);

Serial.println("Arduino Serial.print Function");

Serial.println();

}

void loop(){

Serial.print("NO FORMAT");

Serial.print("\t"); // Друк табуляції

Serial.print("DEC"); // Десяткове

Serial.print("\t");

Serial.print("HEX"); // шістнадцятирічне

Serial.print("\t");

Serial.print("OCT"); // Восьмирічне

Serial.print("\t");

Serial.print("BIN"); // Двійкове

Serial.print("\t");

Serial.println("BYTE"); // Типу Byte

// Розглядається лише частина таблиці ASCII

for(x=0; x< 64; x++){

// Виведення даних в різних форматах

Serial.print(x);

Serial.print("\t");

Serial.print(x, DEC);

Serial.print("\t");

Serial.print(x, HEX);

Serial.print("\t");

Serial.print(x, OCT);

Serial.print("\t");

Serial.print(x, BIN);

Serial.print("\t");

Serial.println(x, BYTE);

delay(200); //Затримка 200 мс

}

Serial.println("");

}

***Serial.println ()***

Функція виводить дані з послідовного порту і здійснює автоматитичний перехід на новий рядок.

***Serial. write ()***

Функція Serial.write() здійснює побайтове виведення даних з буфера послідовного порту:

Serial.write (val);

Serial.write (str);

Serial.write (buf, len);

Значення параметрів:

* val – посилає окремий байт;
* str – посилає рядок побайтово;
* buf – посилає масив побайтово;
* len – довжина масиву ( розмір буфера ).

Лістинг 4.3 ілюструє приклад використання функції Serial.write().

Лістинг 4.3 – Виведення даних у послідовний інтерфейс за допомогою функції Serial.write()

// Arduino Serial.write Funсtion

byte val = 65;

char str[] = "Test";

byte buf[] = {'H','a','l','l','o'};

byte len = 3;

void setup(){

Serial.begin(9600);

Serial.println("Arduino Serial.write Funсtion");

Serial.println();

}

void loop(){

Serial.println("Символ ASCII");

Serial.write(val);

Serial.println();

Serial.println("String 1");

Serial.write(str);

Serial.println();

Serial.println("String 2");

Serial.write(buf, len);

Serial.println();

while(1);

}

У функції Serial.write (buf, len) була вказана змінна len, рівна "3". Таким чином, будуть виведені тільки перші три символи.

**4.1.2 Як функціонує послідовний інтерфейс**

Передача пакета починається зі стартового біта (низький рівень), далі йдуть 8 біт даних. Безпосередньо після бітів даних передається біт перевірки і, нарешті, стоповий біт (рис. 4.2).

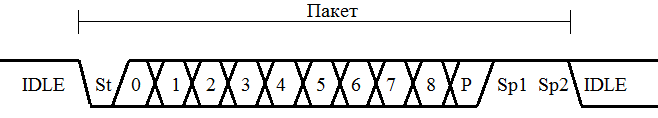


Рисунок 4.2. – Вигляд передачі даних через інтерфейс UART.

St –Стартовий біт, завжди низький рівень;

n – Біти даних (від 0 до 8)

P –біт перевірки

Sp – Стоповий біт, завжди високий рівень

IDLE – Немає передачі по комунікаційній лінії (RxDn або TxDn). При стані IDLE лінія може мати високий рівень

**Зчитування рядка символів** **через послідовний інтерфейс**

Вам вже відомо, як можуть передаватися і прийматися окремі символи і ланцюжки символів (рядки): Для взаємодії і справжнього введення даних слів цього ще не достатньо. Потрібно написати програму, що дозволяє приймати будь-які рядки або блоки даних (лістинг 4.4).

Лістинг 4.4 –Зчитування рядка символів з послідовного інтерфейсу

// Arduino Serial Read

#define INLENGTH 20

#define INTERMINATOR 13

char inString[INLENGTH+1];

int inCount;

void setup(){

Serial.begin(9600);

Serial.println("Arduino Serial read");

Serial.println();

Serial.println("Введіть текст із максимум 20 символів: ");

}

void loop(){

inCount = 0;

do {

while (Serial.available()==0);

inString[inCount] = Serial.read();

if(inString[inCount]==INTERMINATOR) break;

}while(++inCount < INLENGTH);

inString[inCount] = 0;

Serial.print(inString);

}

Програма чекає появи символу в буфері і зчитує його в масив inString. Якщо було прийнято більше 20 символів або символ CR, масив виводиться в термінал.

**Послідовне виведення даних з обчисленням**

Практичний приклад роботи з послідовним інтерфейсом ілюструє лістинг 4.5 – невелика програма, яка перераховує градуси за Цельсієм у градуси за Фаренгейтом і навпаки.

Лістинг 4.5 – Виведення даних з обчисленням

// Arduino Gradus

float Grad = 25.5;

float Fahrenheit = 88.2;

void setup(){

Serial.begin(9600);

Serial.println("Arduino Gradus to Fahrenheit converter ");

Serial.println();

}

void loop(){

Serial.print(Grad);

Serial.print(" Gradus ");

Serial.print(Grad\_to\_Fahrenheit(Grad));

Serial.println(" Fahrenheit");

Serial.println();

Serial.print(Fahrenheit);

Serial.print(" Fahrenheit");

Serial.print(Fahrenheit\_to\_Grad(Fahrenheit));

Serial.println(" Grad");

Serial.println();

while(1);

}

float Grad\_to\_Fahrenheit(float grad){

float erg;

erg = grad \* 9;

erg = erg / 5;

erg = erg + 32;

return erg;

}

float Fahrenheit\_to\_Grad(float fahrenheit){

float erg;

erg = fahrenheit - 32 ;

erg = erg \* 5 ;

erg = erg / 9;

return erg;

}

**4.1.3 Програмна емуляція UART**

Коли виникає необхідність використання декількох послідовних пристроїв, а в мікроконтролері є тільки один апаратний інтерфейс UART, середовище розробки Arduino пропонує можливість програмної емуляції інтерфейсу UART. При програмній реалізації UART використовуються виводи 2 і 3. Дані зчитуються в стек розміром 64 байта. Недолік програмного інтерфейсу UART полягає в тому, що для цього потрібні додаткові системні ресурси.

Обмеження при застосуванні програмного інтерфейсу UART:

* Максимальна швидкість передачі – 9600 бод.
* Відсутня функція serial.available().
* Функція serial.read () чекає до тих пір, поки не з'являться дані в буфері.
* Функція serial.read () повинна запускатися в циклі, якщо функція не викликається, а дані у цей час надходять, вони пропадають.

У бібліотеці UART від Arduino є наступні функції: softwareserial(), begin(), read(), print(), println().

Лістинг 4.6 ілюструє приклад конфігурації послідовного інтерфейсу UART.

Лістинг 4.6 – Приклад програмної реалізації інтерфейсу UART

// Arduino Software UART

#include <SoftwareSerial.h> //Підклювення бібліотеки Software UART

#define rxPin 2

#define txPin 3

#define ledPin 13

// Налаштування програмного UART

SoftwareSerial mySerial = SoftwareSerial(rxPin, txPin);

byte pinState = 0;

void setup() {

// Конфігурація виходів

pinMode(rxPin, INPUT);

pinMode(txPin, OUTPUT);

pinMode(ledPin, OUTPUT);

mySerial.begin(9600);

}

void loop(){

char someChar = mySerial.read(); // Прослуховування порту

mySerial.print(someChar); // Виведення символу на екран

toggle(13); //Зміна стану світлодіода

}

void toggle(int pinNum){

digitalWrite(pinNum, pinState);

pinState = !pinState;

}

**4.2 Конфігурація входу / виходу та настройки порту**

Щоб чіп Arduino знав, який вивод ми хотіли б використовувати як вхід або вихід, потрібно вказати це в підпрограмі void setup(). Далі ми розглянемо, як це зробити. Якщо така конфігурація не виконується, то всі виводи мікроконтролера після включення встановлюються в високоімпедансний стан (Z-стан).

***pinMode (pin, mode)***

Функція використовується в програмі void setup(), щоб конфігурувати контакт плати як вхід або як вихід:

pipМode (pin, OUTPUT); // контакт встановлюється як вихід

У чипі ATmega також є вбудований підтягаючий резистор (20 кОм), який управляється за допомогою програмного забезпечення. До вбудованому підтягуючого резистора можна звертатися в такий спосіб:

pinМode (pin, INPUТ); //Контакт встановлюється як вхід

digitalWrite (pin, HIGH); // Підключається підтягаючий резистор

Підтягаючий резистор зазвичай підключається, щоб приєднувати входи як комутатор. Видно, що контакт конфігурується як вхід, але на ньому встановлюється високий рівень. Це лише метод активізувати внутрішній підтягуючий резистор.

Виводи, що конфігуруються як вихід, мають малий повний опір і можуть навантажуватись приєднаними елементами і схемами з струмом максимум 40 мА. Цього вистачить для свічення світлодіода (з урахуванням послідовно включенного опору), але недостатньо, щоб експлуатувати більшість типів реле, соленоїдів або електродвигунів. Короткі замикання виводів плати Arduino, а також занадто великі струми можуть пошкодити вихідний контакт і навіть вивести з ладу мікроконтролер. Тому краще підключати зовнішні компоненти до виходу через резистор 470 Ом або 1 кОм.

***digitalRead (pin)***

Функція digitalRead () зчитує значення заданого цифрового виводу, з ререзультатом HIGH або LOW, що відповідає 1 або О. Номер виводу може установлюють або як змінна, або як константа (від 0 до 13).

value = digitalRead (Pin); //value присвоюється значення, рівне значенню на вхідному виводі Pin

***digitalWrite (pin, value)***

Встановлює рівень HIGH або LOW на заданому виводі. Номер виводу може задаватися або як змінна, або як константа (від 0 до 13).

digitalWrite (pin, HIGH); // встановлює на контакті високий рівень (+5 В)

**4.3 Зчитування стану кнопки**

**4.3.1 Кнопка з підтягуючим резистором**

У наступному прикладі (лістинг 4.7) зчитується стан кнопки, подключеної до цифрового входу (контакт 12), і результат відображається за допомогою світлодіода. При натисканні на кнопку світлодіод гасне, а при відпусканні – загоряється. Тут справа в тому, що ми активізували внутрішній підтягаючий резистор і, таким чином, наш вхідний вивод 12 має високий рівень. Якщо ми натискаємо на кнопку, яка підключає вивод до GND (загальний провід), світлодіод не горить, оскільки на вході тепер низький рівень. Такого ж результату можна досягти і з використанням зовнішнього підтягуючого резисторв номіналом близько 10 кОм між виводом і джерелом напруги VCC (+5В), при цьому не потрібно примусово встановлювати високий рівень сигналу.

Лістинг 4.7 – Зчитування стану кнопки з підтягуючим резистором

//IO-Pin Tester

int led=13;

int pin=12;

int value=0;

void setup() {

pinMode(led,OUTPUT);

pinMode(pin,INPUT);

digitalWrite(pin, HIGH);

}

void loop() {

value=digitalRead(pin);

digitalWrite(led,value);

}

Вивод контролера ATmega витримує струм до 40 мА. Загальне навантаження на мікроконтролер (в залежності від типу корпусу) не може бути більше 200 мА. Виводи можуть відрізнятися один від одного. Якщо потрібні точні дані, завжди дивіться паспортні дані.

**4.3.2 Кнопка з узгоджуючим резистором**

У попередньому прикладі кнопка була підключена до контакту з внутрішнім підтягуючим резистором. При натисканні кнопки контакт з'єднувався з GND (загальною шиною). У наступному прикладі (лістинг 4.8) показано, як можна опитувати кнопку через напругу живлення VCC (+5 В). Так як тут не включений підтягуючий резистор, то потрібно додати зовнішній узгоджувальний резистор між виводом, до якого підключена кнопка і GND, щоб вивод не опинився в невизначеному стані. Номінал резистора повинен бути близько 10 кОм.

Лістинг 4.8 – Зчитування стану кнопки з узгоджуючим резистором

//Кнопка з узгоджуючим резистором

int led=13;

int pin=12;

int value=0;

void setup(){

pinMode(led,OUTPUT);

pinMode(pin,INPUT);

}

void loop(){

value=digitalRead(pin);

digitalWrite(led,value);

}

**4.4 Введення аналогових даних і АЦП**

Для вимірювання аналогової напруги плата мікроконтролера Arduino має внутрішній Аналого-Цифровий перетворювач (ADC - Analog Digital Converter). Вбудований АЦП 10-розрядний, тобто крок зміни дорівнює значенню аналогового напруги 0,0048 В при опорній напрузі *U*ref = 5 В. Плата мікроконтролера має шість аналогових входів, але тільки один внутрішній АЦП. Втакому разі канали будуть перемикатися, це називають мультиплексна схема. З цієї простої формули можна легко розрахувати роздільну здатність:

*U*step =*U*ref / Розрядність.

*U*step =5 В / 1024 = 0,0048 В

Якщо потрібно дізнатися цифрове значення, то можна розрахувати його за формулою:

Значення = 1024\* (напруга на вході АЦП) / *U*ref

Точність вимірювання складає ± 2 кроки. При опорній напрузі 5 В точність перетворення становить ± 0,0097 В. Можна говорити таким чином, що АЦП вимірює прикладену напругу з точністю до двох цифр після коми. Зрозуміло, що опорна напруга має бути стабільною, оскільки від неї напряму залежить точність вимірювання.

***analogRead (pin)***

Функція зчитує значення встановленого аналогового входу з роздільною здатністю 10 біт. Функція допустима тільки для виводів 0–5 (мається на увазі Arduino Uno):

value = analogRead (pin); // встановлює змінну value дорівнює значенню на напруги на вході

Аналогові виводи, на відміну від цифрових, не потрібно оголошувати як вхід або вихід, на початку програми.

Розглянемо невелику програму (лістинг 4.9) для вимірювання напруги на вході 0 АЦП (ANALOG IN О). для регулювання напруги використовується змінний резистор між виводами A0 і VCC.

Лістинг 4.9 – Зчитування аналогових даних

// ADC

int ADC0=0;

int value;

void setup() {

Serial.begin(9600);

}

void loop() {

value=analogRead(ADC0);

Serial.print("ADC0 = ");

Serial.println(value);

delay(1000);

}

При обертанні ручки потенціометра буде видно, що значення на виході АЦП змінюється від 0 до 1023, в залежності від положення ручки. Максимуму в 1023 можна і не бути досягненим, оскільки деякі потенціометри мають невеликий залишковий опір. Для зручного представлення даних у вольтах, а не в бітах, можна виконати перетворення, додавши до коду вище наступні команди:

float voltage = value \* (5.0 / 1023.0);

Serial.println(voltage);

**4.5 Аналоговий вихід. ШІМ**

На платі Arduino знаходиться шість виходів з широтно-імпульсною модуляцією (ШІМ): контакти 3, 5, 6, 9, 10 і 11. Вони можуть використовуватися для цифроаналогового перетворення, управління сервоприводами або для формування звукових сигналів. В процесі ШІМ (PWM – Pulse Width Modulation) змінюється прогальність імпульсної послідовності. Прогальність вказує співвідношення тривалості включеного стану до періоду повторювання імпульсів. При цьому частота і рівень сигналу залишаються завжди однаковими. Змінюється тільки тривалість переходу від високого до низького рівня (рис. 4.3).

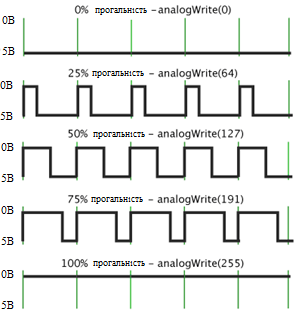
******

Рисунок 4.3 – Ілюстрація прогальності

***analogWrite (pin, value)***

Ця команда формує на виході псевдоаналогові значення за допомогою модуляціі ширини імпульсної послідовності (ШІМ). Значення може встановлюватись у вигляді змінної або константи в діапазоні 0-255:

analogWrite (pin, value); // Запис значення до аналогового виводу

При нульовому значенні напруга на заданому аналоговому виході теж буде нульовою. Значення 255 відповідає напрузі 5 В. При значеннях між 0 і 255 вивод перемикається між 0 і 5 В, чим більше значення, тим довше вивод має високий рівень (5 В). При значенні 64 на виводі три чверті періоду буде 0 В і одну чверть – 5 В. Значення 127 призводить до того, що вихідна напруги одну половину періоду часу має високий рівень і другу – низький рівень. При значенні 191 напруга на виводі протягом однієї чверті періоду становить О В і три чверті – повні 5 В . Так як ця функція базується на апаратних засобах, то постійний сигнал запускається незалежно від програми, і рівень сигналу на виводі буде зберігатись до наступного виклику функції analogWrite() (чи до виклику digitalRead() або digitalWrite() для того ж виводу).

Аналогові виводи, на відміну від цифрових, не потрібно попередньо налаштовувати як вхід або вихід.

У прикладі (лістинг 4.10) два світлодіода перемикається з високою частотою. В результаті значення ШІМ визначає яскравість світіння.

Лістинг 4.10 – Приклад використання ШІМ

// Analog Write

int value;

int LEDgr=10;

int LEDrd=11;

void setup() {

//У даній програмі нічого не потребує попередньої ініціалізації

}

void loop(){

for(value=0;value<255;value++) {

analogWrite(LEDgr, value);

analogWrite(LEDrd, 255-value);

delay(5);

}

delay(1000);

for(value=255;value!=0;value--) {

analogWrite(LEDgr, value);

analogWrite(LEDrd, 255-value);

delay(5);

}

delay(1000);

}

Якщо приєднати невеликий п’єзоакустичний перетворювач до аналогового виходу, то можна "почути" ШІМ-сигнал. Правда, звук буде не дуже гучним.

**4.6 Деякі спеціальні функції**

**Введення паузи за допомогою *delay***

У попередніх прикладах ми вже неодноразово задавали паузу за допомогою функції delay().

***delay (ms)***

Зупиняє програму на час, вказаний в мілісекундах, причому значення 1000 відповідає 1 секунді:

delay (l000); // затримка на одну секунду

***delayMicroseconds()***

Зупиняє програму на час, вказаний в мікросекундах, причому значення 1000 відповідає 1 мілісекунді:

delayMicroseconds (l000); // затримка на одну мілісекунду

**Функції випадкових чисел**

При написанні вимірювальних, керуючих, регулюючих або ігрових програм часто потрібні випадкові числа, наприклад, якщо в будинку в довільний час має запалюватися і гаснути освітлення. Для цієї мети передбачена функція random().

***randomSeed (seed)***

Функція встановлює значення або початкове число для random():

randomSeed (value); // встановлює значення value як випадкове початкове число

Застосування randomSeed () гарантує кращий результат. Випадкові числа можна передати в якості аргументу, наприклад, функцій millis() (див. далі) або analogRead(), щоб згенерувати електричні перешкоди на аналоговому виводі.

***random (min, тах)***

Функція random здійснює генерацію псевдовипадкових значень в межах певного діапазону мінімального і максимального значень:

value = random (l00, 200); // встановлює value випадковим числом в діапазоні між 100 і 200

Використовуйте random (min, max) після функції randomSeed ().

Лістинг 4.11 ілюструє приклад генерації випадкових чисел.

Лістинг 4.11 – Приклад використання функції random

// random numbers

int x,y=0;

void setup(){

randomSeed(100);

Serial.begin(9600);

Serial.println("Arduino random numbers ");

Serial.println();

}

void loop(){

for(x=0;x<20;x++)

{

y=random(0, 10);

Serial.print(y);

Serial.print(",");

}

Serial.println();

for(x=0;x<20;x++){

y=random(10,100);

Serial.print(y);

Serial.print(",");

}

Serial.println();

for(x=0;x<20;x++){

y=random(0,x+1);

Serial.print(y);

Serial.print(",");

}

Serial.println();

while(1);

}

Ось можливі результати виконання лістингу 4.38:

0, 9, 5, 5, 9, 3, 2, l, l, 9, 4, 3, 9, 9, 5, 6, l, 0, 4, 8

83, 24, 24, 99, 92, 36, 97, 35, 13, 10, 43, 98, 88, 52, 89, 86, 29, 35, 37, 58

0, 0, 1, 3, 0, 1, 1, 4, 6, 9, 7,3, 1, 3, 5, 8, 9, 9, 17, 18

Запустіть програму кілька разів і переконайтеся, що завжди видаються різні числа. Але результати завжди будуть знаходитися в заданому діапазоні від мінімального до максимального значення.

**4.7 Вимірювання часових інтервалів**

Щоб визначити, скільки минуло часу після запуску програми або виконання якоїсь операції, в Arduino є спеціальні функції. Зрозуміло, можна самому створювати цикли з зупинкою всієї програми і очікуванням, до тих пір, поки не закінчиться необхідний інтервал часу. Час може визначатися в мілісекундах або мікросекундах.

***millis ()***

Функція повертає значення часу (в мілісекундах), що минув після запуску програми:

value = millis (); // повертає інтервал часу в мілісекундах

Змінна типу unsigned long переповнюється приблизно через 50 днів і обнуляєся. Лістинг 4.12 ілюструє приклад використання цієї функції.

Лістинг 4.12 – Вимірювання часу функцією millis ();

// timekeeping

unsigned long value;

void setup(){

Serial.begin(9600);

Serial.println("Arduino timekeeping

");

Serial.println();

}

void loop(){

Serial.print("Time: ");

value=millis();

Serial.println(value);

delay(1000);

}

***micros ()***

Функція повертає інтервал часу (в мікросекундах) з моменту запуску програми:

value = micros (); // повертає інтервал часу в мікросекундах

Переповнення настає приблизно через 70 хвилин (при тактовій частоті контролера 16 МГц), і підрахунок починається з нуля. Функція використовується аналогічно millis().

1000 мілісекунд відповідають 1000000 мікросекундам.

**ТЕМА 5 ПРОТОКОЛИ ЗВ’ЯЗКУ**

**5.1 Використання протоколу I2C**

**5.1.1 Описінтерфейсу I2C**

Послідовний протокол обміну даними IIC (частіше використовується запис I2C - Inter-Integrated Circuits, міжмікросхемне з'єднання), використовує для передачі даних дві двонаправлені лінії зв'язку, які називаються шина послідовних даних SDA (Serial Data) і шина тактування SCL (Serial Clock) . Також є дві лінії для живлення. Шини SDA і SCL підтягуються до шини живлення через резистори (рис. 5.1).

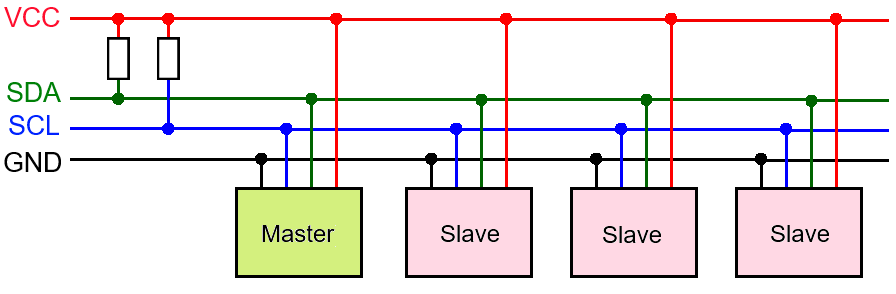


Рисунок 5.1 – Структурна схема мережі I2C

У мережі є хоча б один ведучий пристрій (Master), який ініціалізує передачу даних і генерує сигнали синхронізації. У мережі також є ведомі пристрої (Slave), які передають дані по запиту ведучого. У кожного ведомого пристрою є унікальна адреса, за якою ведучий і звертається до нього. Адреса пристрою вказується в паспорті (datasheet). До однієї шині I2C може бути підключено до 127 пристроїв, в тому числі кілька ведучих. До шині можна підключати пристрої в процесі роботи, тобто вона підтримує «гаряче підключення».

**5.1.2 Реалізація I2Cв Arduino**

Arduino використовує для роботи по інтерфейсу I2C два порти. Наприклад, в Arduino UNO і Arduino NANO аналоговий порт A4 відповідає SDA, аналоговий порт A5 відповідає SCL. Для інших моделей плат відповідність виводів така показана у табл. 5.1

*Таблиця 5.1*

**Реалізація I2C в різних платах Arduino**

|  |  |  |
| --- | --- | --- |
| **плата** | **пін SDA** | **пін SCL** |
| Arduino Uno, Nano, Pro і Pro Mini | A4 | A5 |
| Arduino Mega | 20 | 21 |
| Arduino Leonardo | 2 | 3 |
| Arduino Due | 20, SDA1 | 21, SCL1 |

**Бібліотека "Wire" для роботи з I2C**

Для полегшення обміну даними з пристроями по шині I2C для Arduino написана стандартна бібліотека Wire.h, функції для роботи з якою представлені у табл. 5.2. Більшість пристроїв, котрі підключаються за протоколом I2C для зручності використання мають власні бібліотеки, котрі наслідують функції бібліотеки **Wire.h**

*Таблиця 5.2*

**Функції бібліотеки Wire**

|  |  |
| --- | --- |
| Функція | Призначення |
| **begin(address)** | Ініціалізація бібліотеки та підключення до шини I2C. Якщо не вказана адреса, то пристрій вважається ведучим. |
| **requestFrom()** | Використовується ведучим пристроєм для запиту визначеної кількості байт від ведомого. |
| **beginTransmission(address)** | Початок передачі даних до ведомого пристрою по заданій адресі. |
| **endTransmission()** | Припинення передачі даних ведомому. |
| **write()** | Запис даних від ведомого у відповідь на запит. |
| **available()** | Повертає кількість байт даних, доступних для прийому від ведомого. |
| **read()** | Зчитування байта, переданого між пристроями. |
| **onReceive()** | Вказує на функцію, котра буде викликана при прийомі даних ведомим пристроєм від ведучого. |
| **onRequest()** | Вказує на функцію, котра буде викликана при прийомі даних ведучим пристроєм від ведомого. |

**5.2 Використання протоколу SPI**

**5.2.1 Загальні відомості про протокол SPI**

Інтерфейс SPI, розроблений компанією "Моторола", являє собою повнодуплексний послідовний стандарт зв'язку, який підтримує одночасний двонаправлений обмін даними між ведучим пристроєм і одним або декількома підлеглими. Оскільки протокол SPI не має формального стандарту, робота різних пристроїв SPI може трохи відрізнятися (наприклад, різним може бути число переданих в пакеті біт або може бути відсутнім лінія вибору ведомого пристрою). Далі розглянемо загальноприйняті команди SPI, які підтримуються в Arduino IDE.

Залежно від вимог конкретного пристрою існують чотири основних способи реалізації протоколу SPI. SPI-пристрої виступають при обміні у якості підлеглих синхронних пристроїв, дані синхронізуються з тактовим сигналом (SCLK). Підлеглий пристрій може приймати дані або по позитивному, або по негативному фронту тактового сигналу (так звана фаза синхронізації), а активний стан SCLK за замовчуванням може бути високим або низьким рівнем (так звана полярність синхронізації). У підсумку виходить, що обмін SPI в цілому можна налаштувати чотирма способами (табл. 5.3).

*Таблиця 5.3*

**Режими SPI в Arduino IDE**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Режим SPI | Полярність синхронізації | Фаза синхронізації |
| 1 | Mode 0 | LOW | По фронту синхронізатора |
| 2 | Mode 1 | LOW | По спаду синхронізатора |
| 3 | Mode 2 | HIGH | По спаду синхронізатора |
| 4 | Mode 3 | HIGH | По фронту синхронізатора |

**5.2.2 Підключення пристроїв SPI**

Систему обміну даними через SPI нескладно налаштувати. Для зв'язку між мастером і всіма підлеглими пристроями використовується три виводи:

* послідовний сигнал синхронізації (SCLK);
* вихід ведучого, вхід ведомого (MOSI);
* вхід ведучого, вихід ведомого (MISO).

У кожного ведомого пристрою також є контакт вибору даного пристрою (контакт SS). Отже, загальне число портів введення-виведення, необхідних на мастер-пристрої, завжди буде 3 + *n*, де *n* – число вtдомих пристроїв. Приклад SPI-системи з двома ведомими пристроями зображений на рис. 5.2.

**5.2.3 Конфігурація інтерфейсу SPI**

Будь-який інтерфейс SPI містить, як мінімум, чотири лінії передачі даних. Для кожного ведомого пристрою додаються додаткові лінії SS. Перш ніж відправляти або отримувати дані через SPI, потрібно з'ясувати, що роблять ці лінії введення-виведення і як вони повинні бути підключені (табл.5.4).

*Таблиця 5.4*

**Лінії введення-виведення інтерфейсу SPI**

|  |  |
| --- | --- |
| Лінії SPI | Призначення |
| MOSI | Лінія для відправки послідовних даних від ведучого пристрою до ведомого |
| MISO | Лінія для відправки послідовних даних від ведомого пристрою до ведучого |
| SCLK | Лінія синхронізації послідовних даних |
| SS | Лінія вибору ведомого пристрою, активний рівень – низький |

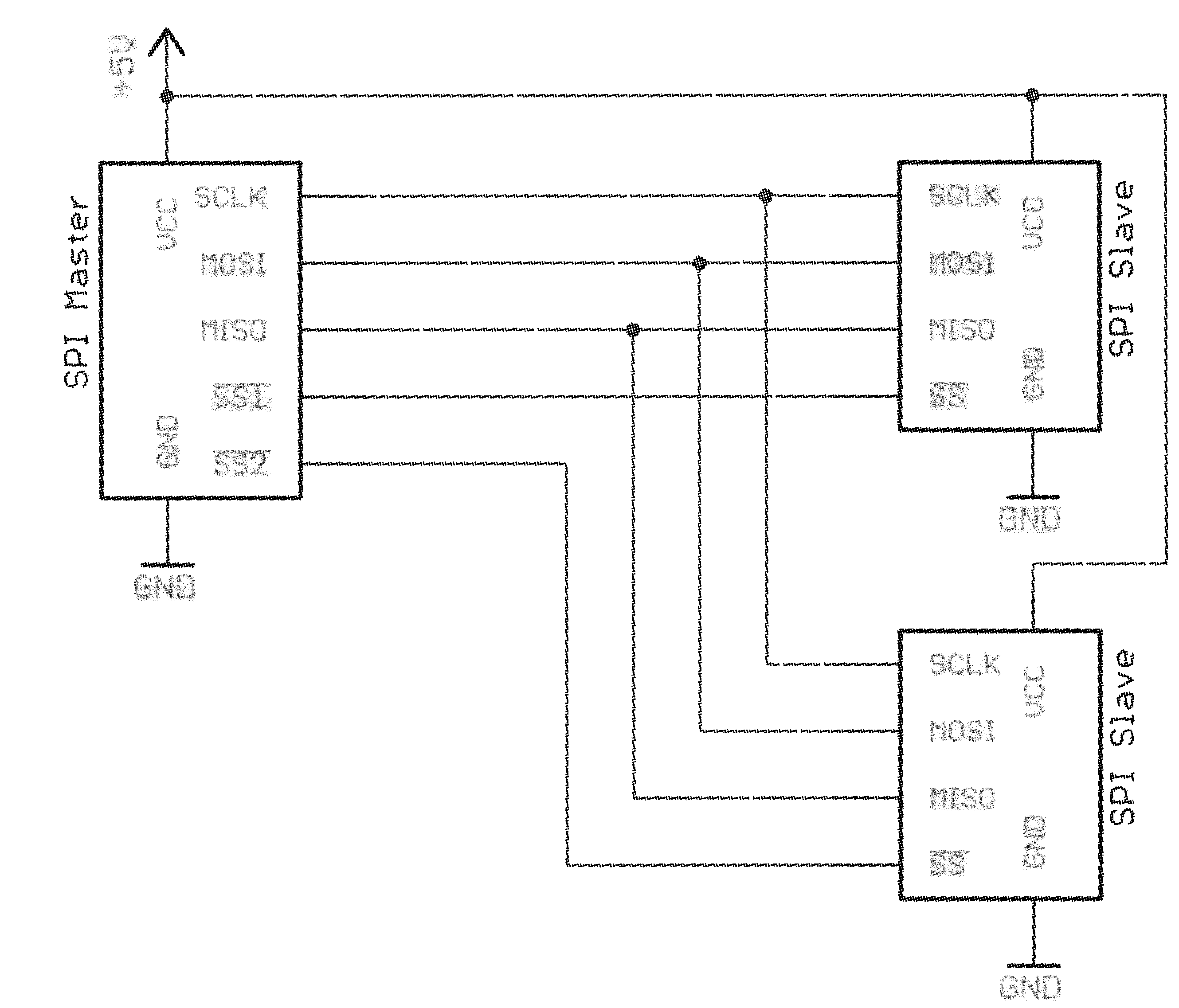


Рисунок 5.2 – Приклад конфігурації SPI-пристроїв

На відміну від інтерфейсу I2С, підтягуючі резистори тут не потрібні, і протокол повністю двонаправлений. Отже, щоб підключити пристрій SPI до плати Arduino, необхідно з'єднати його з контактами MOSI, MISO, SCLK і SS. Після цього все готово до використання Arduino бібліотеки SPI. Так як SPI не є універсальним стандартом, деякі виробники пристроїв SPI можуть по-різному називати лінії зв'язку SPI. Лінію вибору ведомого іноді називають CS, лінію синхронізації - CLK; контакти MOSI і MISO ведомих пристроїв називають входом послідовних даних (SDI) і виходом послідовних даних (SDO) відповідно.

**5.2.4 Протокол передачі даних SPI**

Передача даних по SPI синхронізується тактовим сигналом і залежить від стану ліній SS. Всі команди, що відправляються мастером, проявляються на входах MOSI, MISO, SCLK всіх ведомих пристроїв. Стан контакту SS повідомляє пристрою, ігнорувати ці дані або приймати. При написанні програми слід враховувати, що при передачі даних тільки один контакт SS повинен мати низький рівень.

Послідовність дій для зв'язку з пристроєм SPI виглядає наступним чином:

1. Встановити низький рівень на лінії SS пристрою, з яким хочете встановити зв'язок.

2. Перемикати на тактовій лінії рівень сигналу вгору і вниз зі швидкістю, меншою або рівною швидкості передачі, що підтримується ведомим пристроєм.

3. На кожному такті відправляти по лінії MOSI або отримувати по лінії MISO 1 біт даних.

4. Продовжувати, поки передача (або прийом) не закінчиться, і зупинити перемикання тактової лінії.

5. Установити на SS високий рівень.

**5.2.5 Порівняння SPI і I2C**

На відміну від I2С, шина SPI має окремі лінії для відправки та отримання даних, а також додаткову лінію для вибору ведомого пристрою. Це вимагає наявності додаткових виводів, але усуває проблему адресації ведомого пристрою. SPI-інтерфейс, у порівнянні з I2С, простіший в реалізації і працює на більш високій швидкості. Багато видів пристроїв, в тому числі акселерометри, цифрові потенціометри, дисплеї і т.п., доступні і в SPI- і в I2С-версіях. Зрештою, вибір пристрою залежить від конкретної ситуації. Більшість початківців вважають, що працювати з пристроями SPI легше, ніж з пристроями I2С.

**5.3 Енергонезалежна пам'ять EEPROM**

EEPROM – (Electrically Erasable Programmable Read-Only Memory) постійний запам'ятовувальний пристрій, що програмується та очищується за допомогою електрики, один з видів енергонезалежної пам'яті. Пам'ять такого типу може стиратися і заповнюватися даними кілька десятків тисяч разів. Використовується в твердотільних накопичувачах. Однією з різновидів EEPROM є флеш-пам'ять (Flash Memory).

Мікроконтролери Atmega8 і Atmega168, що працюють в Arduino мають 512 байт EEPROM, Atmega328 - 1кб незалежної пам'яті, в якій можна зберігати будь-які дані, які будуть доступні після відключення живлення. Це може стати в нагоді для зберігання яких-небудь даних або значень.

Для роботи з даною пам'яттю в складі Arduino IDE є бібліотека EEPROM.h (\hardware\libraries\EEPROM\). Бібліотека містить всього дві функції – читання і запису даних.

***byte EEPROM.read (address)***

Функція зчитує байт з незалежної пам'яті EEPROM. Якщо байт до цього ніколи не перезаписувати – повернене значення буде 255.

Параметри:

address – порядковий номер комірки пам'яті для читання - від 0 до 1023 (int) (для Atmega328)

Значення, що повертається: Байт, що зберігається в комірці пам'яті.

Приклад зчитування даних з енергонезалежної пам’яті показаниу у лістингу 5.1.

Лістинг 5.1 – Зчитування даних з EEPROM

//Зчитування EEPROM

//Зчитує значення всіх байт незалежній пам'яті і виведення їх в COM-порт

#include <EEPROM.h>

int address = 0; // початкова адреса пам'яті EEPROM

byte value;

void setup () {

Serial.begin (9600);

}

void loop () {

value = EEPROM.read (address); // зчитуємо значення з поточною адресою EEPROM

Serial.print (address);

Serial.print ( "\ t");

Serial.print (value, DEC);

Serial.println ();

address = address + 1; // встановлюємо наступний елемент пам'яті

if (address == 1024)

address = 0; // якщо адреса досяг 1024, то знову переходимо на 0

delay (500);

}

***Void EEPROM.write (address, value)***

Функція записує байт в енергонезалежну пам'ять.

Параметри:

address: порядковий номер комірки пам'яті для запису - від 0 до 1023 (int)

value: байт для запису - від 0 до 255 (byte)

Значення, що повертається: нічого

Документація (*datasheet*) на мікроконтролери Atmega8 / 168 / 328 говорить, що можлива кількість циклів перезапису даних в пам'яті обмежена 100000 раз (Write / Erase Cycles). Це слід враховувати при використанні даної пам'яті.

Так само документація вказує, що час, необхідний для завершення циклу запису становить 3.3 ms . Якщо в цей час спробувати що-небудь зчитати / записати в EEPROM, то така спроба закінчиться невдачею, однак, дана затримка вже враховується бібліотекою EEPROM.h, тому в додатковому виклику delay () немає необхідності.

Приклад запису інформації в енергонезалежну пам’ять показаний у лістингу 5.2

Лістинг 5.2 – Запис даних в енергонезалежну пам’ять

// Зберігає в незалежній пам'яті EEPROM значення, зчитаного з аналогового входу analog input 0.

#include <EEPROM.h>

int addr = 0; // поточне значення адреси EEPROM

void setup () {

}

void loop () {

int val = analogRead (0) / 4; // ділення на 4 необхідно, щоб перевести значення від 0-1023 у діапазон від 0 до 255.

EEPROM.write (addr, val); // записуємо значення в незалежну пам'ять

addr = addr + 1; // встановлюємо наступний елемент пам'яті.

if (addr == 1024)

addr = 0; //при досягненні // кінця пам'яті - повертаємося на початок

delay (100);

}

**5.4 Використання переривань в Arduino**

Часто при роботі з проектами на мікроконтролерах потрібно запускати фонову функцію через рівні проміжки часу. Це часто реалізується установкою апаратного таймера для генерації переривання. Це переривання запускає програму обробки переривань (Interrupt Service Routine, ISR) для управління періодичним перериванням.

**5.4.1 Поняття переривання**

Переривання (Interrupt) – це сигнали, що переривають нормальний перебіг програми. Переривання зазвичай використовуються для апаратних пристроїв, що вимагають негайної реакції на появу подій. Наприклад, система послідовного порту або UART мікроконтролера повинен бути обслужений при отриманні нового символу. Якщо цього не зробити швидко, новий символ може бути втрачений.

Переривання може бути викликане при настанні якої-небудь події, при цьому виконання поточної послідовності команд призупиняється і керування передається обробнику переривання, який реагує на подію та обслуговує її, після чего повертає управління в перерваний код.

В залежності від джерела виникнення сигналу переривання поділяються на:

* Асинхронні або зовнішні (апаратні) – події, які створені зовнішнімі джерелами (наприклад, периферійними пристроями) та можуть відбутися в довільній момент: сигнал від таймера, мережевої карти або дискового накопичувача, натискання клавіш клавіатури, рух миші;
* Синхронні або внутрішні – події в самому процесорі як результат порушення якихось умов при виконанні машинного коду: поділ на нуль або переповнення, звернення до неприпустимих адрес або неприпустимий код операції;
* Програмні (частковий випадок внутрішнього переривання) – ініціюються виконанням спеціальної інструкції в коді програми. Програмні переривання, як правило використовуються для звернення до функцій вбудованого програмного забезпечення (firmware), драйверів і операційної системи.

При надходженні нового символу UART генерує переривання. Мікроконтролер зупиняє виконання основної програми (вашого додатку) і перескакує на програму обробки переривань (ISR), призначену для даного переривання. В даному випадку це переривання за отриманням символу. Ця ISR захоплює новий символ з UART, поміщає в буфер, потім очищає переривання і виконує повернення. Коли ISR виконує повернення, мікроконтролер повертається в основну програму і продовжує її з точки виклику. Все це відбувається в фоновому режимі і не впливає безпосередньо на основний код вашої програми.

Якщо у вас запускається багато переривань або переривання генерує швидкодіючий таймер, ваша основна програма буде виконуватися повільніше, так як мікроконтролер розподіляє свій машинний час між основною програмою і всіма функціями обробки переривань.

Ви можете задуматися, чому б не просто перевіряти новий символ час від часу, замість використання такого складного процесу переривання. Обчислимо приклад, щоб побачити, наскільки важливі процеси переривання. Скажімо, у вас є послідовний порт зі швидкістю передачі даних 9600 бод. Це означає, що кожен біт символу посилається з частотою 9600 Гц або близько 10 кГц. На кожен біт йде 100 мкс. Близько 10 біт потрібно, щоб послати один символ, так що ми отримуємо один повний символ кожну мілісекунду або близько того. Якщо наш UART буферизується, ми повинні витягти останній символ до завершення прийому наступного, це дає нам на всю роботу 1 мс. Якщо наш UART не буферизується, ми повинні позбутися від символу за 1 біт або 1 мкс. Розглянемо для початку буферізований приклад.

Ми повинні перевіряти отримання байта швидше, ніж кожну мілісекунду, щоб запобігти втраті даних. Стосовно Arduino це означає, що наша функція циклу повинна звертатися для читання статусу UART і можливо, байта даних, 1000 разів в секунду. Це легко здійснимо, але сильно ускладнить код, який вам потрібно написати. Уявіть, що вам потрібно обслуговувати кілька пристроїв введення-виведення, або що необхідно працювати на набагато більшій швидкості передачі. Бачите, які неприємності це може принести.

З перериваннями вам не потрібно відстежувати надходження символу. Апаратура подає сигнал за допомогою переривання, і процесор швидко викличе ISR, щоб вчасно захопити символ. Замість виділення величезної частки процесорного часу на перевірку статусу UART, ви ніколи не повинні перевіряти статус, ви просто встановлюєте апаратне переривання і виконуєте необхідні дії в ISR. Ваша головна програма безпосередньо не зачіпається, і від апаратного пристрою не потрібно особливих можливостей.

**5.4.2 Переривання по таймеру**

Розглянемо використання програмного таймера 2 для періодичних переривань. Вихідна ідея полягала у використанні цього таймера для генерації частоти биття в звукових проектах Arduino. Щоб виводити тон або частоту нам потрібно перемикати порт введення-виведення на узгодженій частоті. Це можна робити з використанням циклів затримки. Це просто, але означає, що наш процесор буде зайнятий, нічого не виконуючи, а чекаючи точного часу перемикати. З використанням переривання по таймеру ми можемо зайнятися іншими справами, а вивод нехай перемикає ISR, коли таймер подасть сигнал, що час настав.

Нам потрібно тільки встановити таймер, щоб подавав сигнал з перериванням в потрібний час. Замість прокрутки циклу для затримки по часу, наша головна програма може робити щось інше, наприклад, контролювати датчик руху або керувати електроприводом. Опишемо ISR тільки в аспектах, що стосуються переривань таймера 2.

**5.4.3 Таймери на Arduino**

Arduino використовує три таймери контролера ATMega.

* Tаймер 0 (Системний час, ШІМ 5 and 6). Використовується для зберігання лічильника часу роботи програми. Функція millis () повертає число мілісекунд з моменту запуску програми, використовуючи ISR глобального збільшення таймера 0. Таймер 0 також використовується для реалізації ШІМ на виводах 5 і 6.
* Tаймер 1 (ШІМ 9 і 10). Використовується для реалізації ШІМ для цифрових виводах 9 і 10.
* Tаймер 2 (ШІМ 3 і 11). Використовується для управління виходами ШІМ для цифрових виводів 3 і 11.

Хоча всі таймери використовуються, тільки Tаймер 0 має призначену таймером ISR. Це означає, що ми можемо використати Таймер 1 і/або Таймер2 під свої потреби. Однак в результаті ви не зможете використовувати ШІМ на деяких портах введення-виведення. Якщо ви плануєте використовувати ШІМ, майте це на увазі. У прикладі використано Таймера 2, що вплине на роботу виводів 3 і 11. Тестова програма повністю відключає ШІМ на цифрових виводах, керованих таймером 2.

**5.4.4 Установка Таймера 2**

Наведена нижче програма (лістинг 5.3) показує створену функцію установки Таймера 2. Ця функція підключає переривання по переповненню Таймера 2, встановлює попередньо заданий масштаб для таймера підраховує завантажуване значення таймера, що дає бажану частоту переривань за часом.

Лістинг 5.3 – Установка таймера 2

#define TIMER\_CLOCK\_FREQ 2000000.0 // 2MHz for / 8 prescale from 16MHz

/\* Установка Таймера2. Конфігурується 8-бітний Таймер2 ATMega для генерації переривання із заданою частотою. Повертає початкове значення таймера, яке повинно бути завантажено в TCNT2 всередині вашої процедури ISR.

\*/

unsigned char SetupTimer2 (float timeoutFrequency) {

unsigned char result; // Початкове значення таймера.

// Підрахунок початкового значення таймера

result = (int) ((256.0- (TIMER\_CLOCK\_FREQ / timeoutFrequency)) + 0.5);

// Установки Таймер2: Дільник частоти / 8, режим 0

// Частота = 16MHz / 8 = 2Mhz або 0.5 мкс

// Дільник / 8 задає робочий діапазон

// Жорстко запрограмуємо це.

TCCR2A = 0;

TCCR2B = 0 << CS22 | 1 << CS21 | 0 << CS20;

// Підключення переривання по переповненню Timer2

TIMSK2 = 1 << TOIE2;

// завантажує таймер для першого циклу

TCNT2 = result;

return (result);

}

Спочатку визначається тактова частота таймера. Показано, що тактова частота встановлена 2 МГц, так як ми використовуємо ділення на 8 (дільник частоти) опорної частоти 16 МГц. Це жорстко запрограмовано в функції. Функція має один аргумент – бажану частоту переривань, і повертає значення, яке необхідно перезавантажувати в таймер в процедурі ISR. Функція не обмежує необхідну частоту, але не слід занадто її завищувати.

Далі підраховується значення, перезавантажуване в таймер. Це дуже простий підрахунок, але вимагає операцій з плаваючою крапкою. На щастя, нам потрібно зробити це тільки один раз, оскільки операції з плаваючою точкою дуже дорого обходяться в перерахунку на машинний час. Приймемо, що таймер буде встановлений на 2 МГц при кожному рахунку. Завантажуване значення – це число відліків, яке ми хочемо зробити при 2 МГц між перериваннями.

Наступна частина коду встановлює таймер в режим 0 і вибирає дільник частоти / 8. Режим 0 – це базовий режим таймера, а дільник / 8 показує, як ми отримуємо лічильник, який рахує за частотою 2 МГц або 0,5 мкс на відлік. Далі підключається переривання по переповненню. Після виконання цього коду мікроконтролер буде викликати ISR кожен раз, коли лічильник прокрутиться від 0xFF до 0 × 00. Це трапиться, коли лічильник прорахує від нашого завантаженого значення через FF і назад до 00. Нарешті, ми завантажуємо значення лічильника в таймер і повертаємо це завантажене значення, щоб ISR могла використовувати його пізніше.

**5.4.5 Завантаження мікроконтролера перериваннями**

Щоб дати вам уявлення про ефект, припустимо, що таймер ISR запускався б кожні 20 мкс. Процесор, що працює на 16 МГц, може виконати близько 1 машинної команди кожні 63 нс або близько 320 машинних команд для кожного циклу переривання (20 мкс). Припустимо також, що виконання кожного рядка програми може зайняти багато машинних команд. Кожна інструкція, яка використовується в ISR, забирає час, доступний для виконання будь-якої іншої програми. Якби наша ISR використовувала близько 150 машинних циклів, ми використали б половину доступного процесорного часу. При активних переривання головна програма відкладалася б на половину часу, займаного нею в інших випадках.

Якщо у вас буде занадто довга ISR, ваша головна програма буде виконуватися вкрай повільно, якщо ж ISR буде довшим, ніж тривалість циклу таймера, то практично ніколи не виконається ваша головна програма, і, крім того, в кінці кінців відбудеться збій системного стека.

**5.4.6 Вимірювання завантаження переривань**

Напишемо тестовий код, який виконує оцінку завантаження і дозволяє вивести вимірювання на послідовний порт. Таймер не встановлено в режим, коли він перезавантажується автоматично. Це означає, що ISR повинна перезавантажити таймер для наступного інтервалу рахунку. Було б правильніше автоматично перезавантажувати таймер, але, використовуючи цей режим, ми можемо виміряти час, що проводиться в ISR, і відповідно виправити час, завантажуваний в таймер. За допомогою цієї корекції ми при розумній точності, також отримуємо і число, що показує, скільки часу ми проводимо в ISR.

Трюк полягає в тому, що таймер зберігає час, навіть якщо він переповнений і перерваний. В кінці нашої ISR ми можемо захопити поточне значення лічильника таймера. Це значення – той час, який він відібрав у нас до наступної точки програми. Це сумарний час, витрачений на перехід в процедуру переривання і виконання програми в ISR. Невелика похибка буде у тому, що не підраховується час, витрачений на команду перезавантаження таймера, але ми можемо виправити її емпірично. Саме тому у формулі підрахунку завантажується значення 257 замість 256, і зайвий такт компенсує команду перезавантаження таймера.

**5.4.7 ISR Таймера2**

ISR для переривання по переповнення Таймера2 показана у лістингу 5.4.

Лістинг 5.4 – Обробка переривань по переповненню таймера 2

#define TOGGLE\_IO 9 // вивод Arduino для перемикання по таймеру ISR

// Timer2 покажчик вектора переривання по переповненню

ISR (TIMER2\_OVF\_vect) {

// Перемикання IO-виведення в інший стан.

digitalWrite (TOGGLE\_IO,! digitalRead (TOGGLE\_IO));

// Захоплення поточного значення таймера. Це величина помилки через затримку обробки переривання і виконання цієї функції

latency = TCNT2;

// Перезавантаження таймера і корекція по затримці

TCNT2 = latency + timerLoadValue;

}

Ця функція коротка і її основне завдання – перемикати порт введення-виведення. Після перемикання вона захоплює поточне значення таймера і використовує його для корекції затримки на перезавантаження таймера. Значення затримки глобальне, його може відстежувати головна програма для вищезазначених вимірювань завантаження. Це число тактів на частоті 2 МГц, яке займає ця ISR для виконання своїх функцій.

Важливо щоб ISR була короткою, так як вона викликається кожні 20 мкс при таймері, налаштованому на 50 кГц. Ви можете робити більше операцій в ISR, але вам потрібно знайти баланс між інтервалом переривань і кількістю операцій, які виконуються в них. Значення затримки допоможе в цьому, як описано нижче.

ISR була б набагато швидше, якби ми безпосередньо зверталися до виводу за допомогою регістрів порту. Але простіше використовувати загальні бібліотечні функції.

**5.4.8 Головна програма. Функція Setup ()**

Функція Setup () (лістинг 5.5) викликається за допомогою системної програми Arduino одноразово при запуску програми. Вона ініціалізує порти введення-виведення і таймер. Вона також виконує виведення в послідовний порт, тобто показує, що програма запущена.

Лістинг 5.5 – Реалізація функції setup () для ініціалізації таймера 2

void setup (void) {

pinMode (TOGGLE\_IO, OUTPUT); // Встановлює порт, який нам потрібно перемикати в ISR, вихідним.

Serial.begin (9600); // Запускаємо послідовний порт

Serial.println ( "Timer2 Test"); // Повідомлення про запуск програми

timerLoadValue = SetupTimer2 (44100); // Запускає таймер і отримує завантажуване значення таймера.

// Виводимо завантажуване значення таймера

Serial.print ( "Timer2 Load:");

Serial.println (timerLoadValue, HEX);

}

Setup () починається з установки порту в режим виходу, так щоб ми могли перемикати його в ISR. Потім вона активує послідовний порт і виводить текстове повідомлення, щоб показати, що програма працює.

Далі викликається функція SetupTimer2 з частотою, встановленою в 44100 Гц, загальною частотою дискретизації звуку. Значення, що повертається зберігається в глобальній змінній timerLoadValue для подальшого використання в ISR.

Нарешті, Setup () виводить timerLoadValue, так що ми можемо переконатися, що воно знаходиться в розумних межах.

На цій стадії таймер запущений і наша процедура ISR викликається із заданою частотою. Якщо підключити осцилограф, ви побачите перемикання виводу, що генерує частоту, рівну половині інтервалу таймера. Половина виходить тому, що ми встановлюємо порт в низький рівень в одній ISR, і записуємо в нього високий рівень в інший.

**5.4.9 Головна програма. функція Loop ()**

Розглянемо програмний код, представлений у лістингу 5.6. Функція циклу викликається циклічно, поки програма запущена. Кожен раз при поверненні з циклу він викликається знову. Текст програми виглядає складним, але насправді все, що він робить – усереднює величину затримки ISR Таймера2 і виводить результати вимірювань після отримання 100 замірів.

Слід зазначити, що функція циклу нічого не робить щодо перемикання лінії введення-виведення. Все це управляється ISR, що дозволяє функції циклу зайнятися іншими речами, не звертаючи уваги на процеси, що відбуваються в ISR. У випадку з послідовним портом, ви не повинні турбуватися про завантаження наступного символу в UART, коли він готовий. Ви займаєтеся своїми справами, а послідовний порт управляється в фоновому режимі. Це одна з переваг програм, керованих перериваннями. Функції викликаються при появі події, і відокремлені від вашої прикладної програми.

Лістинг 5.6 – Реалізація циклу loop () для підрахунку часу виконання обробника переривань

void loop (void) {

// Збирає затримку ISR кожні 10 мс.

delay (10);

// Збирає поточне значення затримки з ISR і збільшує лічильник на 1

latencySum + = latency;

sampleCount ++;

// Як тільки набереться 10 замірів, обчислює і виводить результат вимірювань

if (sampleCount> 99) {

float latencyAverage;

float loadPercent;

// Обчислює середню затримку

latencyAverage = latencySum / 100.0;

// обнуляє значення лічильника

sampleCount = 0;

latencySum = 0;

// Обчислює очікуваний відсоток завантаження процесора

loadPercent = latencyAverage / (float) timerLoadValue;

loadPercent \* = 100; // Перераховує частки у відсотки;

// Виводить середню затримку

Serial.print ( "Latency Average:");

Serial.print ((int) latencyAverage);

Serial.print ( ".");

latencyAverage - = (int) latencyAverage;

Serial.print ((int) (latencyAverage \* 100));

// Виводить очікуваний відсоток завантаження

Serial.print ( "Load:");

Serial.print ((int) loadPercent);

Serial.println ( "%");

}

}

Функція циклу починається з затримки 10 мс. (Зауважте, що ми не можемо використовувати подібні затримки, якщо потрібно перемикати порт введення-виведення з високою частотою без переривань). Затримка 10 мс регулює те, як часто ми робимо заміри затримки і виведення вимірювань. Після того як ми отримаємо 100 вимірювань, кожне з яких займає 10 мс, результат виводиться кожну секунду.

Далі значення часу затримки з ISR накопичується в глобальній змінній latencySum. Ми просто захоплюємо поточне значення затримки і додаємо його до того, що вже накопичено. Крім того, ми збільшуємо на одиницю лічильник накопиченого кількості вимірів.

Тепер ми перевіряємо, чи було вже накопичено 100 замірів. Якщо ні, то пропускаємо решту коду і повертаємося. Якщо накопичилося 100 замірів, то розраховуємо середнє значення і зберігаємо результат в latencyAverage. Після цього очищаємо акумулятор і лічильник вимірів, щоб можна було почати все заново.

Тепер, маючи вимір затримки, ми можемо підрахувати очікуване завантаження процесора. Нам відомо, що таймер переповнюється кожен раз, коли він дораховує від нашого перезавантажуваного значення до 0xFF і потім назад до 0х00, це показник переповнення. Відсоток завантаження буде визначатися як затримка / число тактів в ISR. Це значення підраховується і виводиться як результат вимірювання, так що ми можемо оцінити вплив нашої програми ISR.

Час, який займає головна програма – це залишок часу від виконання всіх фонових ISR, ISR цього таймера і інших, постійно активних, наприклад, Таймера0 і послідовного порту.

У прикладі програми таймер завантажений шістнадцятковим значенням D4, або десятковим 212. Це означає, що він буде перерваний кожен раз, як тільки відлічить 44 такти. Ми знаємо, що поки процесор виконує код ISR, таймер відраховує близько 20 тактів, так що залишається близько 24 до того, як він знову повернеться в ISR. Ці 24 такти – весь час, який наша головна програма отримує на виконання. Так що із загального часу 44 такти між перериваннями ми витрачаємо 20 тактів на ISR, залишаючи близько 24 тактів прикладній програмі. Це становить близько 45%, витрачених процесором на ISR.

Сама ISR, без обслуговування будь-яких корисних завдань, використовує близько 6% доступних ресурсів. Час, що залишився використовують функції цифрового читання і запису.