НАВЧАЛЬНО-МЕТОДИЧНА ЛІТЕРАТУРА

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ
УНІВЕРСИТЕТ ІМЕНІ ІВАНА ПУЛЮЯ

Кафедра приладів та контрольно-вимірювальних систем

Написання програм мовою С для мікроконтролерів MCS51

М Е Т О Д И Ч Н І В К А З І В К И
до лабораторних робіт

з дисципліни

Проектування вбудованих систем

ТЕРНОПІЛЬ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

Кафедра приладів та контрольно-вимірювальних систем

М Е Т О Д И Ч Н І В К А З І В К И
до практичних робіт з

з дисципліни

Проектування вбудованих систем

ТЕРНОПІЛЬ 2021

Методичні вказівки до практичних робіт з дисципліни “Проектування вбудованих систем” для студентів спеціальності 8.05100306 «Інформаційні технології в приладобудуванні». / Уклад.: А. В. Чайковський. – Тернопіль: ТНТУ 2021– 41 с.

Призначені підготовки студентів до практичних робіт із дисципліни “Проектування вбудованих систем”. Складається з урахуванням модульної системи навчання, тем практичних занять, типової форми та вимог для комплексної перевірки знань з дисципліни.

Розглянуто на засіданні
кафедри приладів та контрольно-вимірювальних систем

протокол №___ від _____2021р.

Затверджено на засіданні методичної комісії факультету контрольно-вимірювальних та радіокомп’ютерних систем

протокол №___ від _____2021р.

ВСТУП

Методичні вказівки до практичних робіт з дисципліни «Проектування вбудованих систем» розроблені відповідно до навчального плану та робочої програми дисципліни і призначені для студентів спеціальності 8.05100306 «Інформаційні технології в приладобудуванні» освітньо-кваліфікаційного рівня «магістр».

Метою вивчення дисципліни "Проектування вбудованих систем" є освоїти методи проектування локальних вимірювально-обчислювальних систем. Завдання: вивчити структуру типових вбудованих систем, методи їх проектування та програмування.

У результаті вивчення навчальної дисципліни студент повинен

знати:

– архітектуру мікроконтролерів MCS51;

– методи взаємодії із типовою периферією;

– основні структурні блоки мови С;

– принципи написання програмного забезпечення для вбудованих систем;

– принципи побудови вимірювально-обчислювальних систем на основі мікроконтролерів;

вміти:

– проектувати вбудовані системи на основі мікроконтролерів;

– взаємодіяти із периферією мікроконтролерів;

– використовувати мову С для створення програмного забезпечення мікроконтролерних вимірювально-обчислювальних систем.

Таблиця 2.1 – Перелік тем лабораторних робіт

№	Тема	Об'єм в год.	Примітка
ЛАБОРАТОРНІ ЗАНЯТТЯ			
МОДУЛЬ 1.			
1	Використання операторів та функцій.	2	Лабораторна робота №1
2	Використання змінних, констант та масивів.	2	Лабораторна робота №2
3	Використання операторів розгалуження програми.	2	Лабораторна робота №3
МОДУЛЬ 2.			
4	Використання ітеративних алгоритмів.	2	Лабораторна робота №4
5	Керування кроковими двигунами.	2	Лабораторна робота №5
6	Використання АЦП ADuC841.	2	Лабораторна робота №6
Всього за семестр		12	

ЛАБОРАТОРНА РОБОТА № 1

Тема. Використання операторів та функції в програмах написаних мовою С.
Мета. Освоїти принципи написання програм для мікроконтролерів мовою С. Ознайомитися із передачею паралельною шиною даних.
Матеріал. Функції, оператори, коментарі, директива #include, адресація стенду.

а. Короткі теоретичні відомості

Мова С

Загальні відомості

Мова С – мова програмування високого рівня розроблена на початку 70-х Кеном Томпсоном та Денісом Рітчі, Bell Labs. Її основними перевагами в порівняні із асемблером є зручність написання, відлагодження і підтримки великих програм та можливість легкого перенесення коду на інший вид контролерів.

Опис та застосування функцій

Програма, написана мовою С складається із набору функції, що викликають одна одну. Функція визначається таким чином:

```
<тип> <назва>([параметри])
{
  <тіло>
}
```

<тип> тип результату, що повертає функція. Ключове слово **`void`** означає, що функція не повертає результату. Результат повертається за допомогою оператора **`return.`**

<назва> назва функції. Ідентифікатор, що використовується для виклику функції. Ідентифікатор може складатися із послідовності великих чи маленьких латинських букв, цифр та знаку підкреслення. Першим символом не може бути цифра. С розрізняє великі та малі букви (тобто sin та Sin не одне й теж).

<параметри> список даних, які отримує функція. Параметри розділяються комами. Тип параметру вказується перед назвою. Функція може не приймати жодного параметру.

<тіло> послідовність операторів, які реалізують функцію. Оператор від оператора відділяється символом “крапка з комою”.

Яким чином відбувається виклик функції? Для виклику функції компілятором використовує інструкцію контролера **`call`**, виконання функції завершується інструкцією **`ret`**.

Як відбувається передача параметрів у функцію? Компілятор Keil намагається розмістити параметри в регістрах R5-R7. Перед викликом функції компілятор вставляє інструкції, що завантажують дані в регістри. Якщо це можливо, результат повертається через регістри R5-R7. Якщо параметри чи результат не вміщаються в регістрах, то Keil використовує пам’ять даних.

Приклад 1. Функція `a2`, яка приймає один цілий параметр і повертає цілий результат.

```
char a2(char a)
{
  return a*a;
}
```

Приклад 2. Функція `send`, яка приймає два байти адреси та один байт даних і нічого не повертає.

```
void send(int addr, char value)
{
  P0 = addr;
  P1 = addr >> 8;
  P2 = value;
  WR = 1;
  WR = 0;
}
```

Для того, щоб викликати функцію потрібно вказати її ім’я, а далі, в дужках, дані, що передаються функції. Дужки обов’язкові навіть якщо функція не приймає жодного параметру. Наприклад:

```
init();
b = a2(a);
send(1, b);
```

Реалізація функції не обов'язково має передувати її виклику і може бути розміщена будь-де в програмі. Проте, щоб компілятор зміг згенерувати коректний виклик функції, йому потрібна інформація про параметри, що мають передаватися у функцію. Тому першому використанню функції має передувати принаймні оголошення її прототипу. Оголошення прототипу виглядає майже так само як і визначення функції і складається із заголовка функції (тип, назва, параметри) і закінчується символом крапки з комою. Тіло відсутнє, а назви параметрів можуть бути упущені.

```
<тип> <назва>([параметри]);
```

В наступному прикладі функція `x2()` реалізована після функції `main()`, яка її викликає. Звертання до `x2()` можливе через те, що її прототип оголошений вище `main()`.

```
int x2(int x);//Прототип функції
int x2(int);  //Назви параметрів можуть бути упущені

void main()
{
  ...
  x=x2(x);   //Виклик функції
  ...
}

int x2(int x)//Реалізація функції
{
  return x*x
}
```

Часто прототипи функцій виносять в окремий файл або на початок тексту програми. Оголошувати прототип функції необов'язково, якщо всі звертання до неї розміщені нижче опису функції.

Старт програми

Виконання програми написаної мовою С розпочинається із функції main().

```
void main()
{
}
```

Зазвичай для програм мікроконтролерів, функція main() після ініціалізації входить в безкінечний цикл. Тіло циклу може, наприклад, опитувати клавіатуру, обновляти індикатори чи опрацьовувати прапорці, встановлені процедурами обробки переривань. Для реалізації безкінечного циклу можемо скористатися оператором **while**.

```
void main()
{
  //Ініціалізація програми
  while (1)
  {
    //Тіло циклу
  }
}
```

Оператори

В С існують унарні, бінарні та тринарні оператори (виконують операції відповідно над одним, двома та трьома операндами). Послідовність виконання операторів можна змінити використовуючи дужки (). Таблиця 1 коротко описує оператори мови С.

Таблиця 1 – Оператори мови С

Оператори	Оп	Приклад	Дія
Арифметичні			
-	1	-a	Повертає від'ємне значення a
+ - * / %	2	a+b a%2	Повертають відповідно суму, різницю, добуток, частку та остачу від ділення a і b
++a	1	++a b=++a	Префіксний інкремент. Спочатку збільшує на одиницю a і повертає результат
a++	1	a++ b=a++	Постфіксний інкремент. Повертає значення a і після цього збільшує на одиницю a.
--a	1	--a	Префіксний декремент. Спочатку зменшує на одиницю a і повертає результат
a--	1	a--	Постфіксний декремент. Повертає значення a і після цього зменшує на одиницю a.
Логічні			
!	1	!a	Повертає логічне заперечення a
> < >= <= != ==	2	a>b, a!=b	Порівнює операнди a і b і повертає не нуль якщо умова виконується інакше повертає нуль. != означає "не рівне"
&& ||	2	a && b a || b	Повертають відповідно результати логічної операції «і» та «або»
Бітові			
~	1	~a	Повертає побітове заперечення a
& | ^	2	a & b a^0x0F	Повертають відповідно результати побітової операції «і», «або» чи «виключне або»
<< >>	2	a<<b a>>8	Зсуває a на b біт вліво чи вправо і повертає результат
Присвоєння			
=	2	a = b a = b = c	Заносить значення a в b та повертає b
+= -= /= *= %= &= |= ^=	2	a+=b a+=4 a&=0x0f	Виконує відповідну операцію на операндах a і b, заносить результат в a і повертає його. a+=b еквівалентне a=a+b
Адресні			
&	1	ptr = &a	Повертає адресу змінної
*	1	a = *ptr	Непряме звертання до пам'яті
sizeof()	1	sizeof(a) sizeof(int)	Повертає розмір змінної чи типу в байтах
Умовний			
?:	3	max = (a<=b)?b:a	op1 ? op2 : op3 Якщо op1 рівний нулю, то повертається op3 інакше – op2

Константи

В мові С існує чотири типи констант: цілі числа, числа із плаваючою крапкою, символи та стрічки.

Цілою константою є число, записане в десятковій, вісімковій чи шістнадцятковій системі. Запис десяткової константи складається із десяткових цифр, перша з яких не рівна нулю. Наприклад 0, 1, 15, 255. Якщо запис числа розпочинається з нуля, то компілятор сприймає його як вісімкове. Таке число може складатися із цифр 0..7. Наприклад: 0, 01, 017, 0377. Запис шіснадцяткової константи розпочинається із послідовності 0x або 0X та складається із шістнадцяткових цифр 0-9,A,-F. Можна застосовувати і малі букви a-f. Наприклад: 0x0, 0x1, 0xf, 0xFF.

Запис константи із плаваючою крапкою складається із десяткових цифр цілої та дробової частини розділених крапкою та (або) показника експоненти записаного після символу E. Наприклад: 0.0, -1.0, 3.14, 1.25E-3, 12E6.
Символьною константою є один символ оточений з обох сторін апострофами. Значення символьної константи – код символу. Наприклад: ' ' (пробіл), 'a' (символ a), '0' (символ 0).
Стрічковою константою є послідовність символів оточених лапками. Наприклад: "Ethernet", "Результат вимірювання:", "C = 0x". Для того, щоб об'єднати стрічки достатньо записати їх через пробіл. Таким чином можна записувати стрічкову константу в декількох рядках.

Директива компілятора #include

Прототипи функцій і оголошення глобальних змінних в мові С прийнято зберігати в файлах-заголовках із розширенням .h. До основної програми вони підключаються за допомогою директиви компілятора #include. Таким чином можна розділювати програму на окремі звершені модулі.
Існує два варіанти застосування директиви `#include`: `#include <ім'я файлу>` та `#include "ім'я файлу"`. Перший вказує шукати файл у стандартних директоріях операційної системи. Другий – в поточному каталозі. Наприклад:

```
#include <ADUC841.H>
#include <intrins.h>
#include "stend.h"
```

Опис регістрів спеціальних функцій мікроконтролера ADuC841 міститься у файлі ADUC841.H. Тому для звертання до периферії контролера слід підключити його за допомогою директиви

```
#include <ADUC841.H>
```

Коментарі

Мова С передбачає використання коментарів двох типів. Для того, щоб наказати компілятору проігнорувати послідовність від поточної позиції до кінця стрічки можна скористатись двома косими рисками: //. Щоб вилучити із процесу компіляції довільну послідовність символів слід заключити її між символами `/*` та `*/`. Коментарі не можна вкладати. Наприклад:

```
a = 2; //Це коментар
/*b = a+3;
Закоментований довільний відтинок програми*/
```

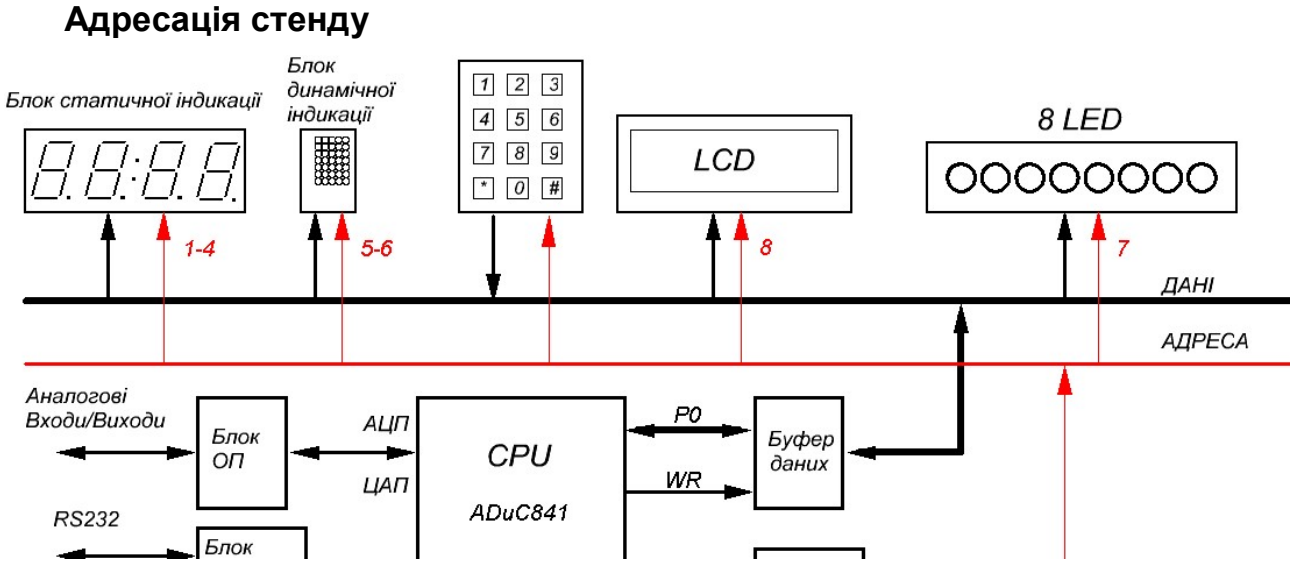


Рисунок 1 – Адресація стенду

Схема адресації стенду зображена на рисунку 1. До восьмибітної шини даних одночасно під'єднані декілька пристроїв: регістри статичного індикатора, матричного дисплею, лінійки світлодіодів, крокових двигунів, LCD дисплей, клавіатура. Однак зміна рівнів на шині даних не вплине на стан вищезгаданих регістрів доти, доки на вході WR відповідного регістру не буде переходу з нульового в одиничний рівень. Іншими словами запис в регістр відбувається по передньому фронту сигналу WR регістра (рисунок 2). Таким чином керуючи сигналами WR ми можемо змінити стан будь-якого регістру не впливаючи на інші.

Доступ до шини даних здійснюється за допомогою порту 0 до якого під'єднаний двонаправлений буфер даних. Сигнал WR (P3.6) задає напрямок передачі буферу даних: 1 – буфер працює на передачу, 0 – буфер працює на прийом. Для коректного прочитання стану лінії даних перед встановленням сигналу WR в 0 слід налаштувати порт P0 на прийом. Для цього в регістр спеціальних функцій P0 слід записати значення 0xff. Для того, щоб змінити стан лінії даних слід встановити сигнал WR (P3.6) в 1 і занести потрібні дані в РСФ P0.

Записом в регістри керує дешифратор адреси під'єднаний до молодшої тетради порту 2 (P2.0-P2.3). Виходи дешифратора CS1..CS15 під'єднані до входів WR регістрів. Вибір адреси 1..15 приводить до встановлення рівня логічного 0 на відповідному провіднику CS1..CS15. Вибір адреси "0" приводить до того, що на всіх лініях CS1..CS15 встановлюється рівень логічної 1. Регістр фіксує дані D0..D7 при переході сигналу запису WR з рівня логічного 0 в рівень логічної 1. Фіксовані дані визначають стан виходів Q0..Q7 при низькому рівні сигналу на вході дозволу EZ (на стенді вихід завжди дозволений).

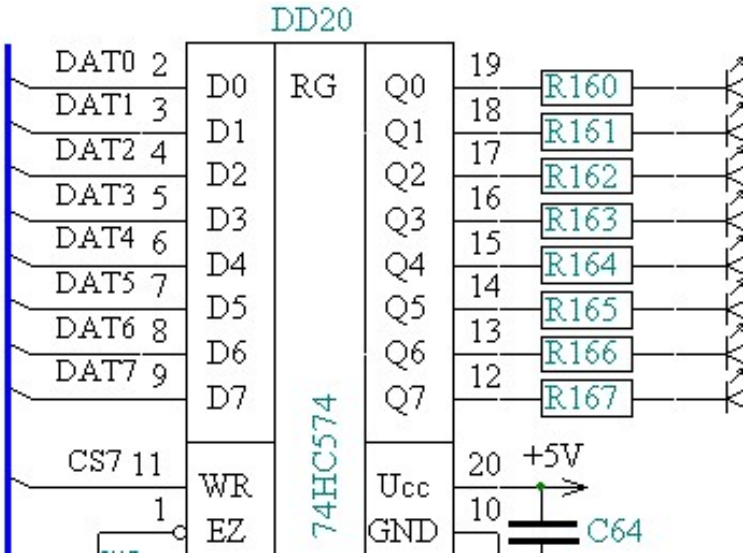


Рисунок 2 – Регістр 574. Світлодіодна лінійка.

Наприклад, щоб записати в регістр стану лінійки світлодіодів байт 0x7A потрібно згенерувати таку послідовність сигналів (рисунок 3).

```
WR = 1;     //0: Переключити буфер даних на запис.
P0 = 0x7A; //1: Виставити дані на шину
P2 = 7;     //2: Записати дані – встановити лінію CS7 в "0"
P2 = 0;     //3: Встановити лінію CS7 в "1". Дані записані
```

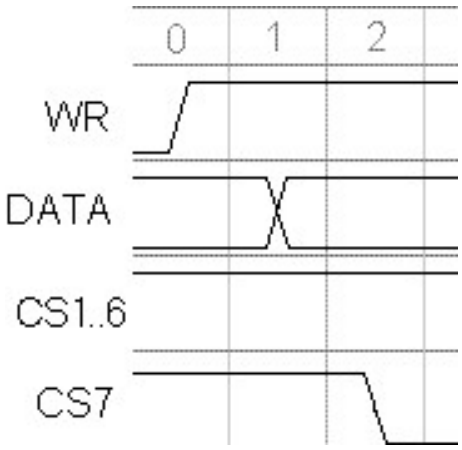


Рисунок 3 – Запис даних в регістр

Недоліком попереднього прикладу є те, що при записі адреси змінюється стан старшої тетради порту P2. Щоб обійти це, можемо скористатися із побітових операцій. Удосконалений метод запису виглядатиме таким чином.

```
WR = 1;      //0: Переключити буфер даних на запис.
P0 = 0x7A;  //1: Виставити дані на шину
P2 &= 0xf0; //Очистка молодшої тетради порту 2
P2 |= 7;     //2: Побітове або.
P2 &= 0xf0; //3: Встановити лінію CS7 в "1". Дані записані
```

Практична частина

1. В середовищі Keil створіть новий проект для мікроконтролера ADuC841. Налаштуйте параметри завантаження. Створіть та додайте до проекту файл main.c.
2. У файлі main.c реалізуйте такі функції

```
void write(unsigned char Addr,
unsigned char Data)
```
Запис байту Data в регістр Addr

`void LED(unsigned char Data)`	Відображення байту Data на лінійці світлодіодів (1 – вмикає світлодіод)
`void ClearLatches()`	Очистка регістрів 1..8 (функція записує байти 0xFF)

3. Реалізуйте функцію main(), яка очистить регістри, відобразить на лінійці світлодіодів байт 0xAA, а далі виконуватиме порожній цикл.
4. Відкомпілюйте та завантажте програму. Перевірте її роботу. Виправте помилки.

Контрольні питання

1. Що означає ключове слово **`void`**?
2. Які обов'язкові елементи опису функції?
3. Яким чином викликати функцію?
4. Яким чином викликати ще нереалізовану функцію?
5. Яким чином описується прототип функції?
6. Як впливають розриви рядків та додаткові пробіли в тексті програми на результат?
7. Як впливає використання великих та малих літер в програмі написаною мовою С?
8. Які вимоги ставляться до ідентифікатора?
9. Яким чином можна вказати компілятору проігнорувати фрагмент тексту?
10. З чого розпочинається виконання програми?
11. Для чого служить головний цикл програми?
12. Які групи операторів ви знаєте?
13. Що таке директива компілятору?
14. Для чого служить директива `#include`?
15. Як записати дані в регістр 74HC574?
16. Знайдіть помилку в коді:

a.

```
void main();
{
};
```

b.

```
void init()
{
}
void main()
{
  init;
}
```

c.

```
include <ADUC841.H>
```

ЛАБОРАТОРНА РОБОТА № 2

Тема. Змінні, константи та масиви.

Мета. Вивчити типи даних та навчитися використовувати змінні та константи.
Ознайомитися із схемою статичної індикації на семисигментному індикаторі.

Матеріал. Опис і використання змінних, констант, вказування типу пам'яті, масиви. Семисегментний індикатор.

Короткі теоретичні відомості

Опис змінних на мові С

Змінна мовою С описується таким чином

```
<const> <тип> <тип пам'яті> <назва>;
```

<назва> будь-який, ще не використаний ідентифікатор.

<тип> тип змінної, що вказує на її розмір і вміст. Стандартні типи змінних наведені в таблиці 1.

<тип пам'яті> необов'язковий параметр, що вказує компілятору де саме розміщувати змінну. Типи пам'яті, що підтримує компілятор Keil наведені в таблиці 2. Якщо тип пам'яті не вказаний, то компілятор намагатиметься спершу розмістити змінну в регістрах, а якщо це не вдалося – то у внутрішній пам'яті даних (**data**).

<**const**> необов'язкове ключове слово, що означає опис константу. Змінити її значення в програмі неможливо. Тип пам'яті **code**, автоматично означає опис константи.

Таблиця 1 – Базові типи (Keil C51)

Базовий тип	Назва	Розмір, біт	Діапазон
signed char, char	Однобайтна ціла зі знаком	8	-128..127
unsigned char	Однобайтна ціла без знаку	8	0..255
signed int, int	Двобайтна ціла зі знаком	16	-32768..32767
unsigned int	Двобайтна ціла без знаку	16	0..65535
signed long	Чотирибайтна ціла зі знаком	32	-2147483648.. 2147483647
unsigned long	Чотирибайтна ціла без знаку	32	0.. 4294967295
float	Число з плаваючою крапкою	32	1.5E-45..3.4E38 7-8 значущих цифр
bit	Біт контролера 8051	1	0..1

Таблиця 2 – Типи пам'яті

Тип пам'яті	Назва
code	Пам'ять програм. Лише константи. Звернення через MOVC @A+DPTR
data	Прямоадресована внутрішня пам'ять (0..127). Найшвидший доступ.
idata	Внутрішня пам'ять із непрямою адресацією. Доступ до всіх комірок 0..255
bdata	Внутрішня пам'ять із можливістю побітової адресації.
xdata	Зовнішня пам'ять. Доступ по MOVX @DPTR
far	Розширена зовнішня пам'ять (адреси до 16МБайт).
pdata	Зовнішня пам'ять із посторінковою адресацією MOVX @Rn

Мінімальний опис змінної складається з її типу та назви. Наприклад:

```
unsigned char i;
```

Якщо описати змінну поза межами функцій, то вона буде **глобальною** – звертатися до неї зможе будь-яка функція. **Локальна** змінна видима тільки в межах блоку, в якому її задекларовано. Так, щоб обмежити область видимості змінної тільки для однієї функції потрібно описати її на початку її тіла. В наступному прикладі константа `a` – глобальна і видима для кожної функції. Змінні `b` та `c` – локальні і доступні лише в межах функції `main()`.

```
const unsigned char code a = 5;
void main()
{
  unsigned char b;
  int c;
  c = b*a;
}
```

Зразу ж після оголошення вміст локальних змінних невизначений. Ініціалізацію змінної можна виконати зразу ж після її оголошення. Значення констант можна задати лише таким способом. Наприклад:

```
unsigned char a=0,b=0;
const int Pi=314;
```

Масив – послідовність змінних одного типу. Для опису масиву після назви змінної вказують його розмір.

```
<базовий тип> [тип пам'яті] <назва>[<розмір>]
```

Для ініціалізації масиву констант потрібно після його опису у фігурних дужках через кому перелічити рівно n значень елементів масиву.

```
const <базовий тип> [тип пам'яті] <назва>[n] = {значення 0, значення 1,
..., значення n-1}
```

Наприклад:

```
unsigned char D[16];  //Масив із 16 змінних типу unsigned char
const float code coefs[3] = {0.1, 0.25, -0.01}; //Масив із трьох
                                                //констант
```

Для ініціалізацію масиву символів можна використати стрічкову константу. Врахуйте, що розмір масиву буде на 1 елемент більший кількості символів в стрічці – компілятор автоматично додає нульовий байт – символ закінчення стрічки.

```
unsigned char code msg[]="Hello world!";
```

Якщо розмір масиву не вказується, то він приймається рівним кількості елементів ініціалізації. Наприклад наступний рядок є коректним оголошенням масиву із трьох елементів:

```
int A[]={1, 2, 3};
```

Звернутися до окремого елементу можна вказавши в квадратних дужках індекс елементу (масив нумерується з нуля). Наприклад:

```
int D[3];
int i=0;
D[0] = 1;
D[1] = D[0]+1;
D[i] = D[i+1]*i;
```

Семисегментний індикатор

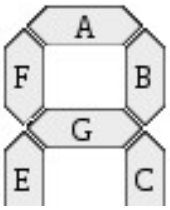


Рисунок 1 – Семисегментний індикатор

В навчальному стенді доступ до семисегментних індикаторів здійснюється шляхом запису байту стану в регістри з адресами 1..4. При цьому «0» засвічує сегмент. Відповідність біт та сегментів така: $PGFEDCBA_2$ (Рисунок 1). Наприклад, щоб засвітити цифру 7 в найстаршій позиції потрібно записати за адресою 4 число $11111000_2= F8_{16}$.

Отже, щоб відобразити шістнадцяткову цифру потрібно перевести її в семисегментний код. Для цього можемо скористатися масивом констант, який слугуватиме таблицею перекодування.

```
unsigned char code LEDD[16] =
  {0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
   0x80, 0x98, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E};
```

Для відображення двобайтного числа послідовно запишемо його 4 тетради, що відповідають чотирьом шістнадцятковим цифрам числа від 0000_{16} до $FFFF_{16}$, у відповідні позиції статичного індикатора. Для відображення цифр перетворимо тетради в семисегментний код.

```
void Static(unsigned int A)
{
  write(1, LEDD[A & 0xF]);
  write(2, LEDD[(A >> 4) & 0xF]);
  write(3, LEDD[(A >> 8) & 0xF]);
  write(4, LEDD[(A >> 12) & 0xF]);
}
```

Практична частина

1. Відкрийте проект, створений на попередньому занятті.
2. Доповніть його такими функціями

`void Static(unsigned int A)`	Відображення на індикаторі двобайтного шістнатцяткового числа
`void StaticL(unsigned char A)`	Відображення в молодших розрядах індикатора однобайтного шіснатцяткового числа
`void StaticH(unsigned char A)`	Відображення в старших розрядах індикатора однобайтного шіснатцяткового числа
`void ClearStatic()`	Очистка індикатора (запис у регістри 1..4 `0xFF`)

3. Функцію `main()` змініть таким чином, щоб після старту програми вона очистила регістри та відобразила на статичному індикаторі число `0x1234`.
4. Відкомпілюйте та завантажте програму. Перевірте правильність її роботи.

Контрольні питання

1. Які основні типи даних підтримує мова С?
2. В чому різниця між цілими із знаком та без знаку?

3. В чому різниця між змінними та константами?
4. Як описати масив констант?
5. Знайдіть помилки в коді:
   a.
   ```
   unsigned char code x;
   x = 0;
   ```
   b.
   ```
   unsigned const char a = 5;
   Static(a);
   ```
   c.
   ```
   float a[3];
   int count;
   a[3] = 0;
   ```

ЛАБОРАТОРНА РОБОТА № 3

Тема. Розгалуження програми.

Мета. Навчитися створювати розгалуження в програмі.
Навчитися використовувати дискретні кнопки та матричну клавіатуру.

Матеріал. Складений оператор, умовний оператор, оператор **switch**, матрична клавіатура та дискретні кнопки.

Короткі теоретичні відомості

Умовний оператор

Умовний оператор застосовується для створення розгалужень програми. Загальний його вигляд такий:

```
if (умова) оператор1; [else оператор2;]
```

Його виконання розпочинається із оцінки умови. Якщо результат обчислення умови ненульовий, то виконується оператор1, інакше виконується оператор2. Наприклад:

```
if (a>b)
  max = a;
else
  max = b;
```

Зверніть увагу! Типовою помилкою є використання оператора присвоєння = замість оператора порівняння == в умові. Наприклад, вираз **if** (x=0) є цілком правильним з точки зору синтаксису С, але замість очікуваної дії порівняння змінної x з нулем виконається операція присвоєння, а вираз поверне нульове значення. Для порівняння слід застосувати оператор ==

```
if (x==0)
  ...
```

Умовні оператори допускається вкладати. Для спрощення прочитання програми та уникнення невизначеності користуйтеся складеним оператором. Наприклад:

```
if (key!=255)
{
  if (key==0)
  {
    ProcessKey(Readkey());
  }
  else
    ProcessKey(key);
};
```

Складений оператор

Складений оператор застосовується у випадку якщо потрібно виконати декілька операторів там, де синтаксис С передбачає лише один. Загальний вигляд складеного оператора такий

```
{
  [оголошення змінних, констант, типів]
  ...
  [оператор1];
  [оператор2];
  ...
  [операторN];
}
```

Після складеного оператора крапка з комою ставити не потрібно. На відміну від мови Паскаль в С крапка з комою обов'язкова також і після останнього оператора.

Змінні, оголошені на початку складеного оператора, доступні лише в його межах. Так в наступному прикладі змінна c, доступна лише в межах складеного оператора {…}, а змінна і має інше значення а ніж в межах основної програми (тобто приховує змінну головної програми).

```
void main()
{
  int a, b;
  char i;
  i = 10;
  if (i!=0)
  {
    int c, i;
    c = 0; //Тут змінна с доступна,
    i = 0; //а під і компілятор  розуміє іншу змінну
    ...
  }
  a = i;   //Тут змінна с не доступна, а і знову має значення 10
};
```

Оператор switch

Оператор **switch** використовується, якщо потрібно реалізувати розгалуження на декілька гілок. Його загальний вигляд такий:

```
switch (вираз)
{
  case константний_вираз1:
    оператор1_1;
    ...
    оператор1_N;
    break;
   ...
  case константний_виразК:
    операторК_1;
    ...
    операторК_N;
    break;
  default:
    оператор_1;
    ..
    оператор_N;
}
```

Вираз, записаний в дужках після ключового слова **switch** може бути довільним виразом, який повертає ціле значення. Тіло оператора **switch** складається із декількох наборів операторів помічених ключовим словом **case** разом із константним виразом. Всі константні вирази в межах одного **switch** мають бути унікальними. Необов'язкова секція **default** виконується якщо не виконалася жодна попередня. Для того, щоб перервати послідовність виконання служить ключове слово **break**. Приклад використання оператора **switch**

```
Key = Readkey();
switch (Key)
{
  case 10:
    GoBack();
  break;
  case 11:
```

```
    Execute();
  break;
  default:
    ProcessKey(Key);
}
```

Виконання оператора **switch** відбувається таким чином.

1. Оцінюється значення виразу, записаного у фігурних дужках після ключового слова switch.
2. Отриманий результат послідовно порівнюється із константними виразами.
3. Якщо результат співпадає з константним виразом, то програма продовжує виконання із наступного оператора.
4. Після цього оператори виконуються один за одним доти, доки не зустрінеться ключове слово **break**, яке завершує виконання switch.
5. Якщо результат не співпав із жодним константним виразом, то виконуються оператори, помічені ключовим словом **default**.

Дискретні кнопки стенду

Дві дискретні кнопки стенду суміщені з механічним енкодером і джойстиком та приєднані до ліній P3.2 (INT0) та P3.3 (INT1) відповідно. Відпрацювати їх натиск можна шляхом опитування бітів INT0 та INT1 або обробником переривання. Наприклад опитування можна реалізувати таким чином:

```
if (INT0)
  LED(0);
else
  LED(1);
```

Матрична клавіатура стенду

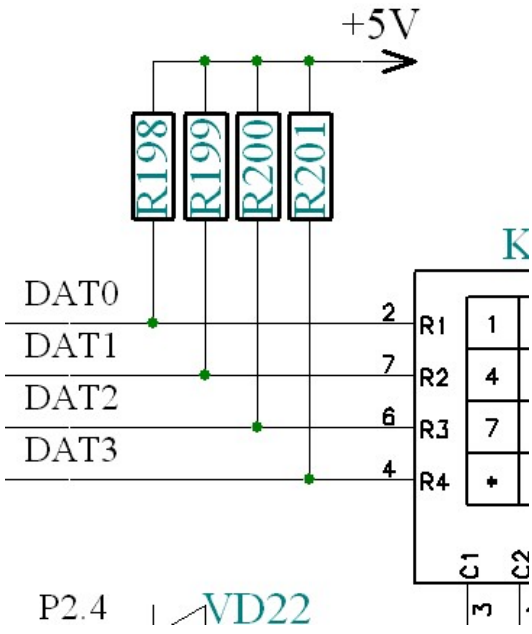


Рисунок 1 – Матрична клавіатура

Стовпці матричної клавіатури стенду приєднані до ліній P2.4-P2.6, рядки – до чотирьох молодших ліній шини даних. Для опитування клавіатури потрібно перевести порт 0 та буфер даних в режим зчитування, а потім почергово встановлювати низький рівень сигналу на лініях P2.4-P2.6 та зчитувати стан клавіш відповідного стовпця з шини даних (порту 0). Приклад опитування першого стовпця:

```
unsigned char ReadKey()//Опитування першого стовпця клавіатури
{
  P0 = 0xFF;    //Переключити порт 0 на прийом
  WR = 0;       //Переключити буфер даних на прийом
  P2 |= 0x70;   //Заповнити P2.4-P2.6 одиницями
  P2 &= ~0x10; //Встановити P2.4 в нульовий стан
  switch (P0|0xF0)//Дешифрація клавіші (старші біти встановлюємо в 1)
  {
    case ~1: return 1;  //Нуль в першому рядку — натиснута кнопка 1
    case ~2: return 4;  //Нуль в другому рядку — натиснута кнопка 4
    case ~4: return 7;  //Нуль в третьому рядку — натиснута кнопка 7
    case ~8: return 10; //Нуль в четвертому рядку — натиснута кнопка *
  }
  return 255;  //Якщо не натиснена жодна клавіша
}
```

Для опрацювання всієї клавіатури попередній алгоритм слід розширити, додавши опитування другого та третього стовпців. Таким способом можна відпрацювати натиск єдиної клавіші.

Практична частина

Перший рівень

1. Створіть копію проекту Stend, створеного на попередніх практичних заняттях. Відкрийте її і розширте файл main.c функцією опитування матричної клавіатури

`unsigned char ReadKey()`	повертає код натисненої клавіші або 255, якщо жодна клавіша не натиснена. Код клавіш "0".."9" – 0..9; клавіші "*" – 10; клавіші "#" – 11.

2. Функцію main() змініть таким чином щоб після старту програма очищала індикацію і розпочинала виконувати головний цикл.
3. Головний цикл
   - опитує кнопку енкодера і відображає на лінійці світлодіодів 1, якщо вона натиснута і 0 в іншому разі;
   - опитує клавіатуру та відображає код клавіші на статичному індикаторі
4. Відкомпілюйте та завантажте програму. Перевірте правильність її роботи.

Другий рівень

5. Змініть основну програму так, щоб код натиснутої клавіші продовжував відображатися після відпускання.

Третій рівень

6. Змініть основну програму так, щоб індикатор відображав число без знаку яке б збільшувалось на одиницю при натисканні клавіші "#" і зменшувалося на одиницю при натисканні клавіші "*".

Контрольні питання

1. Для чого використовується складений оператор?
2. Як записати мовою С складений оператор?
3. Як описати умовний оператор?
4. Як виконати більше ніж один оператор в кожній із двох гілок розгалуження?
5. Для чого застосовується оператор **switch**?

6. Якого типу може бути ключовий вираз в дужках після оператора **switch**?
7. Яке значення ключового слова **default**?
8. Для чого використовується оператор **break**?
9. Опишіть алгоритм опитування матричної клавіатури.
10. Знайдіть і виправте помилки в коді:

a.

```
if (a>max) then
  ...
```

b.

```
if (a=0)
  a = 255;
```

c.

```
if (a>b)
  max = a
else
  max = b;
```

d.

```
switch (key)
{
  case 10: Volume++;
  case 11: Volume--;
}
```

ЛАБОРАТОРНА РОБОТА № 4

Тема: Цикли в мові С.

Мета: Навчитися використовувати цикли для програмування повторюваних операцій.
Ознайомитися із принципами керування кроковими двигунами.

Матеріал: Цикл for, while, do while, оператори break та continue, крановий двигун.

Короткі теоретичні відомості

Цикли використовуються для програмування багаторазового повторення однієї і тієї ж послідовності операцій. Організувати цикл мовою С можна за допомогою оператора **`for`**, **`while`**, **`do`**…**`while`**. Багаторазове повторення певної дії також може бути запрограмоване використовуючи рекурсію чи оператор **`goto`**.

Якому способу надати перевагу? Використання **`goto`** не рекомендується через утруднене читання програми. Рекурсія, хоч і елегантний, проте ресурсоємний спосіб який вимагає динамічного виділення пам'яті, а тому для програм мікроконтролерів вона малопридатна. Серед решти операторів циклів слід обрати той, за допомогою якого програмована дія буде записана найпростіше, найзрозуміліше.

Цикл for

Цикл **`for`** складається із заголовка та тіла. Заголовок розпочинається ключовим словом **`for`** і містить три частини записані в дужках і розділені символом крапки з комою. Перша частина виконується найпершою один раз і найчастіше використовується для ініціалізації змінних циклу. Друга частина – умова – перевіряється *перед* кожною ітерацією циклу. Якщо її значення не рівне нулю (істина), то виконується тіло циклу, інакше виконання циклу завершується. Якщо умова хибна ще перед першою ітерацією, то тіло циклу не виконається жодного разу. Третя частина – оператор реініціалізації – виконується щоразу *після* виконання тіла і найчастіше використовується для модифікації змінної циклу.

Тілом вважається наступний після заголовка оператор. Якщо в циклі потрібно повторювати більше ніж одну операцію, то слід скористатися складеним оператором {}. Якщо всі необхідні дії виконує заголовок, то тіло можна залишити порожнім. В такому випадку після заголовку відразу ж ставлять крапку з комою – порожній оператор.

Розглянемо цикл **`for`** на прикладах. Наступний фрагмент викличе функцію `write` вісім разів. При цьому змінна `i` набуватиме значення від 1 до 8. Після останньої ітерації, коли змінна `i` стане рівна 9, а умова, записана в другій частині заголовку (`i!=9`), стане хибною (поверне 0) виконання циклу перерветься.

```
for (i=1; i!=9; i++)
  write(i, 0xff);
```

Будь-яка частина заголовку може бути порожньою. В такому випадку, для виходу із циклу потрібно в тілі додатково передбачити виклик оператора **break**. Наприклад:

```
for (;;)
{
  ...
  if (a>b) break;
  ...
}
```

Кожна частина заголовку може містити і більше ніж один оператор – їх можна записати через кому. Наприклад дзеркальне відображення масиву можна реалізувати так:

```
for (i=0,j=last;i<=j;i++,j--)
{
  t = a[i];
  a[i] =  a[j];
  a[j] = t;
}
```

Як приклад циклу із порожнім тілом розглянемо фрагмент пошуку першого нульового елементу масиву:

```
for (j=0; a[j]!=0; j++);
```

Цикл while

Цикл **while** або цикл з передумовою зручно використовувати у випадку якщо наперед невідома кількість повторень. Заголовок циклу з передумовою складається із ключового слова while і умови записаної в дужках. Умова перевіряється *перед* кожною ітерацією. Якщо вона істинна, то виконується тіло циклу, інакше цикл переривається. Тіло циклу теж може бути складеним чи порожнім оператором. Наприклад, наступний код інтегрує вхідну напругу доти, доки сума не перевищить значення 32768:

```
while (sum<32768)
  sum += GetADC(0);
```

Цей же цикл можна було б записати інакше, але зрозумілість коду втрачається:

```
while ((sum+=GetADC(0))<32768);
```

Завжди істинна умова часто використовується для організації основного циклу програми. Якщо ж такий "вічний" цикл потрібно завершити, то можна скористатися оператором **break**. Наприклад:

```
while (1)
{
  ...
    if (errCode!=0) break;
  ...
}
```

Цикл із порожнім тілом всі необхідні дії виконує в заголовку. Наприклад наступний фрагмент очікуватиме одиничного рівня на лінії T0:

```
while (!T0);
```

Цикл do-while

На відміну від циклів **for** та **while**, цикл **do**...**while** перевіряє умову виходу *після* кожної ітерації. Це означає, що такий цикл виконається хоча б раз за будь-яких умов. Опис оператора розпочинається ключовим словом **do**, далі

записується один або декілька операторів тіла циклу, а завершується опис ключовим словом **while** разом із умовою записаною в дужках після нього. Після кожного проходу перевіряється умова і якщо вона істинна, то тіло виконується ще раз, інакше цикл переривається.
Наступний приклад очікує натиску клавіші і записує її код в змінну Key:

```
do
{
  Key = ReadKey();
}
while (Key!=255)
```

Оператори break та continue

Оператори **break** та **continue** використовуються для керування ходом виконання циклу в його тілі. Оператор **break** перериває поточну ітерацію і завершує виконання циклу. Оператор **continue** – перериває поточну ітерацію і розпочинає наступну. Для циклів **while** та **do**...**while** **continue** означає негайний перехід до перевірки умови виходу. Для циклу **for** перед перевіркою умови виходу виконується оператор реініціалізації.
Наприклад для того, щоб знайти суму всіх невід'ємних елементів масиву можна скористатися таким фрагментом коду:

```
for (i=0; i!=last; i++)
{
  if (a[i]<0) continue;
  s += a[i];
}
```

Цю ж дію можна запрограмувати так:

```
for (i=0; i!=last; i++)
  if (a[i]>=0)
    s += a[i];
```

Вкладені цикли

Тіло циклу можуть складати різні оператори також і цикли. Це надає можливість створювати вкладені цикли. Наступний код знаходить суму елементів двохвимірного масиву:

```
unsigned char A[5][3];
unsigned char i, j;
unsigned int  Sum=0;
for (i=0; i<5; i++)
  for (j=0; j<3; j++)
    Sum += A[i][j];
```

Керування кроковими двигунами

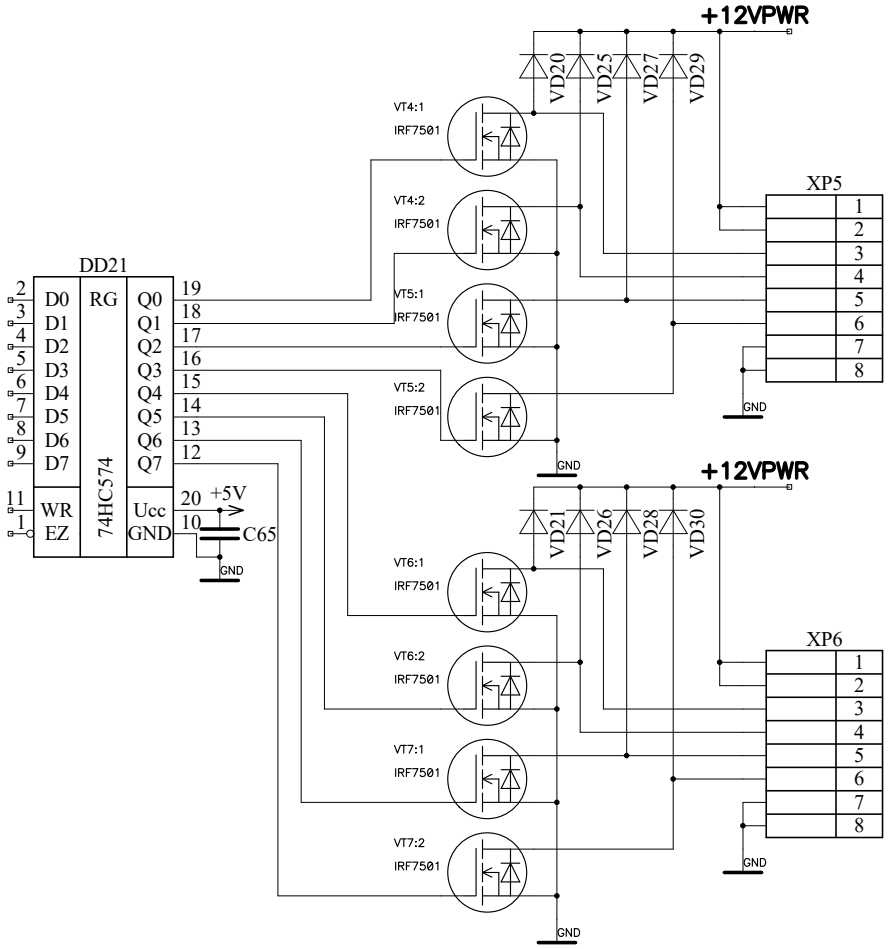


Рисунок 1 – Ключі двигунів

Для керування уніполярними кроковими двигунами на стенді передбачено додатковий регістр, виходи якого комутують 8 ключів типу IRF7501. Діоди, увімкнені між стоками (drains) і додатною напругою живлення в зворотному напрямку необхідні для захисту ключів від напруги само-е.р.с. яка виникає в обмотках двигуна під час її відключення.

Керування кроковим двигуном здійснюється шляхом почергового ввімкнення його обмоток. Для кожного двигуна існує гранична частота перемикання перевищення якої приведе до пропускання кроків. Гранична частота обертання залежить від моменту опору двигуна та режиму роботи.

В повнокроковому режимі (full stepping) за один крок вмикається одна обмотка. Кількість кроків в цьому режимі рівна паспортній, а споживаний струм – найменший.

Таблиця 3 – Послідовність ввімкнення обмоток в повнокроковому режимі

№ кроку	1	2	3	4
Стан обмотки 1	1	0	0	0
Стан обмотки 2	0	1	0	0
Стан обмотки 3	0	0	1	0
Стан обмотки 4	0	0	0	1

В півкроковому режимі (half stepping) спочатку вмикається перша обмотка, далі – перша і друга, далі – друга, далі – друга і третя і так далі. Кількість кроків в цьому режимі вдвічі більша від паспортної, споживаний струм теж більший. Завдяки більш плавній роботі вища гранична частота обертання.

Таблиця 4 – Послідовність комутацій обмоток в півкроковому режимі

№ кроку	1	2	3	4	5	6	7	8
Стан обмотки 1	1	1						1
Стан обмотки 2		1	1	1				
Стан обмотки 3				1	1	1		
Стан обмотки 4						1	1	1

В мікрокроковому режимі для досягнення ще більшої роздільної здатності регулюється струм, що подається на кожну з обмоток. Для цього можна скористатися наприклад широтно-імпульсною модуляцією.

Таблиця 5 – Послідовність комутацій обмоток в мікрокроковому режимі

№ кроку	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Обмотка 1	1	1	1	0.5										0.5	1	1
Обмотка 2		0.5	1	1	1	1	1	0.5								
Обмотка 3						0.5	1	1	1	1	1	0.5				
Обмотка 4										0.5	1	1	1	1	1	0.5

Практична частина

Перший рівень

1. Створіть копію папки проекту. Назвіть її lab5. Відкрийте проект і доповніть його такими функціями:

`void ClearLatches()`	Використовуючи цикл for, записує в регістри 1..8 байт 0xff.
`void Delay(unsigned int T)`	Створює затримку, пропорційну до параметру T.
`unsigned char Step(bit Reverse)`	Якщо параметр Reverse рівний 0 то обертає кроковий двигун на півкроку вперед, інакше – півкроку назавд. Функція повертає стан обвиток двигуна (одиничні біти – обвитка заживлена).
`void StepV(signed char V)`	Якщо V>0 робить крок вперед, якщо V<0 робить крок назад, інакше не змінює положення. Індикує стан обвиток на лінійці світлодіодів а швидкість на статичному індикаторі. Виконує затримку на час зворотньопропорційний до модуля V.

2. Функцію `main()` змініть так, щоб після старту програми двигун обертався вперед із сталою швидкістю. Стан обмоток і швидкість відображайте відповідно на лінійці світлодіодів та статичному індикаторі.
3. Відкомпілюйте та завантажте програму. Перевірте правильність її роботи.

Другий рівень

1. Функцію `main()` змініть так, щоб після старту програми швидкість обертання двигуна циклічно змінювалася відповідно до варіанту. Стан обмоток і швидкість відображайте відповідно на лінійці світлодіодів та статичному індикаторі.

Варіант	Завдання
0	Плавний розгін, плавний вибіг
1	Плавний розгін, стала швидкість, вибіг.
2	Плавний розгін вперед, плавний вибіг, плавний розгін назад, плавний вибіг.
3	Плавний розгін, плавний вибіг, зупинка
4	50 кроків вперед, 50 кроків назад

Контрольні питання

1. Якими способами можна запрограмувати багаторазове повторення операторів?
2. Чим загрожує використання оператора **goto**?
3. З чого складається опис циклу **while**?
4. В якій послідовності виконується цикл **while**?
5. В чому різниця між циклами **while** та **do while**?
6. З яких частин складається заголовок оператора **for**?
7. Яким чином можна вказати декілька операторів в кожній частині заголовку циклу **for**?
8. В якій послідовності виконується цикл **for**?
9. Знайдіть і виправте помилки в коді:

a.

```
while (a>b) do
{
 ...
}
```

b.

```
while (sum<5);
  sum += GetADC();
```

c.

```
while ()
{
...
}
```

d.

```
for (i=0; i++; i!=8)
  write(i, 0xff);
```

e.

```
while (sum<10)
{
  Static(sum);
}
```

ЛАБОРАТОРНА РОБОТА № 5

Тема. Аналогово-цифровий перетворювач.
Мета. Ознайомитися із методами підключення АЦП. Освоїти АЦП ADuC841.
Матеріал. Регістри АЦП. Рекомендації щодо роботи з АЦП

Короткі теоретичні відомості

В мікроконтролер вбудоване дванадцятирозрядне АЦП послідовних наближень. За допомогою мультиплексора на його вхід можна подати сигнал із одного з 8 аналогових входів, давача температури, двох вбудованих цифро-аналогових перетворювачів, напруги нуля чи опори. Пристрій вибірки-зберігання дозволяє перетворювати швидкозмінні сигнали без помилок. Кожне перетворення складається з двох фаз. Впродовж першої відбувається захоплення сигналу – конденсатор пам'яті 32пФ (рисунок 1) за допомогою ключів SW1, SW2 під'єднується до джерела і заряджається до вхідної напруги через резистор 200Ом. Чим більший вихідний опір джерела сигналу тим більше часу потрібно для цього. Тому до аналогового входу варто підключити RC ланку, яка служитиме акумулятором заряду для конденсатора пристрою вибірки та зберігання (рисунок 2).
Під час другої фази ключі SW1, SW2 перемикаються в режим зберігання а захоплений заряд зрівноважується цифро-аналоговим перетворювачем. Отриманий при цьому код зберігається в регістрах ADCDATAH:ADCDATAL. Щоб виключити саморозряд конденсатора важливо, щоб тривалість перетворення була досить короткою. Тому тактова частота АЦП не може бути меншою 400кГц.

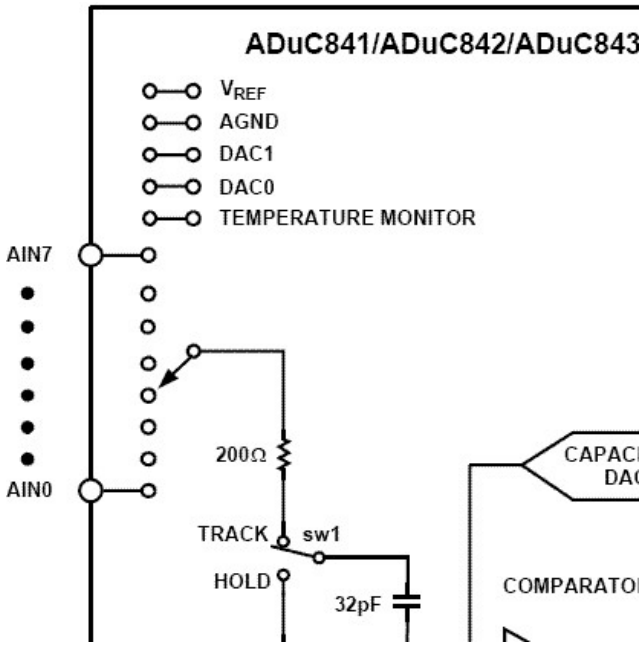


Рисунок 1 – Спрощена схема АЦП мікроконтролера ADuC841

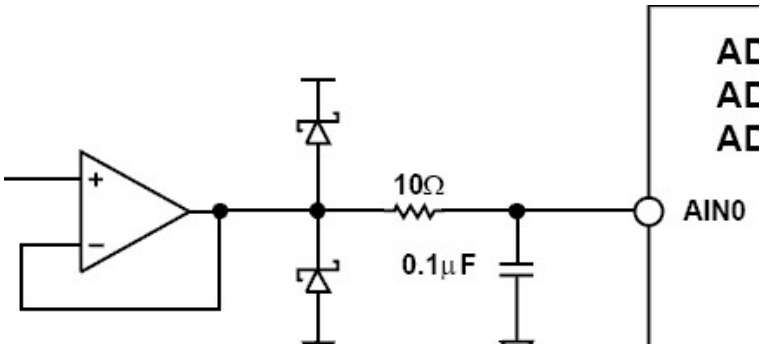


Рисунок 2 – Підключення аналогового сигналу

АЦП керується за допомогою регістрів спеціальних функцій ADCCON1..ADCCON3. Молодших 8 біт результату перетворення зберігаються в регістрі ADCDATAL, старших 4 біти – в молодшій тетраді регістру ADCDATAH. В старшу тетраду контролер записує номер поточного каналу.
Значення регістру ADCCON1 керує режимом запуску та роботи АЦП. Його адреса – 0xEF, початкове значення – 0x40, побітова адресація не підтримується.

Таблиця 1 – Біти регістру ADCCON1

Біт	Назва	Призначення
7	MD1	Ввімкнення АЦП. Якщо біт встановлений – АЦП ввімкнено
6	EXT_REF	Біт встановлений – АЦП працює від зовнішнього джерела опорної напруги, Біт скинений – АЦП працює від внутрішнього джерела опорної напруги.
5	CK1	Визначають подільник тактової частоти в частоту АЦП. Частота АЦП не може
4	CK0	перевищувати 8.38MHz
		CK1 CK0 Подільник
		0 0 32
		0 1 4
		1 0 8
		1 1 2
3	AQ1	Визначають кількість періодів АЦП, що витрачаються на захоплення сигналу
2	AQ0	пристроєм вибірки та зберігання (апертурний час). Для високоомних джерел сигналу значення мають збільшуватися.
		AQ1 AQ0 Кількість періодів
		0 0 1
		0 1 2
		1 0 3
		1 1 4
1	T2C	Встановлений біт дозволяє запуск АЦП при переповненні таймера 2
0	EXC	Встановлений біт дозволяє запуск АЦП при низькому рівні на лініій CONVST

Молодша тетрада регістру ADCCON2 визначає обраний канал перетворення, старша керує запуском АЦП. Адреса регістру – 0xD8, початкове значення – 0x00, побітова адресація підтримується.

Таблиця 2 – Біти регістру ADCCON2

Біт	Назва	Призначення
7	ADCI	Переривання АЦП. Встановлюється контролером вкінці єдиного АЦП перетворення або вкінці блоку перетворень в режимі DMA
6	DMA	Вмикає режим DMA
5	CCONV	Запускає режим послідовних перетворень. Після закінчення кожного перетворення розпочинається нове, доти, доки не користувач не скине біт DMA.
4	SCONV	Запускає одне перетворення. Після його завершення біт автоматично скидається
3	CS3	Визначає активний канал АЦП

2	CS2	CS3	CS2	CS1	CS0	Канал
1	CS1	0	0	0	0	ADC0
0	CS0	.....				
		0	1	1	1	ADC7
		1	0	0	0	Датчик температури (не менше 1мкс для захоплення)
		1	0	0	1	DAC0 (ЦАП 0)
		1	0	1	0	DAC1 (ЦАП 1)
		1	0	1	1	AGND (аналоговий спільний)
		1	1	0	0	VREF (опорна напруга)

Регістр ADCCON3 керує процедурою калібрування і не розглядається в даній лабораторній роботі.

Опитування АЦП

Перед тим, як запускати аналогово-цифрове перетворення слід ввімкнути АЦП – записати в регістр ADCCON1 байт ініціалізації. Наприклад ввімкнемо АЦП з такими параметрами:
Опора – внутрішня (EXT_REF=0)
Подільник – 2 (CK1=1 CK0=1) (fADC=fCLK/2=5.5299МГц)
Апертура – 4 періоди (AQ1=1 AQ0=1)
Запуск від спрацювання таймера 2 та зовнішнього сигналу заборонені (T2C=0 EXC=0)
Таким чином байт ініціалізації рівний $10111100_2=188_{10}=BC_{16.}$ Запишемо його в регістр ADCCON1:

```
ADCCON1 = 0xBC;
```

Далі, щоб провести одиничне вимірювання, потрібно записати в ADCCON2 номер каналу і біти запуску. Так для запуску одиничного перетворення з 7 каналу в ADCCON2 слід записати байт 0x17.
Після запуску дані будуть готові через 17-20 тактів АЦП (в залежності від апертурного часу, встановленого бітами AQ1-AQ0). Якщо подільник частоти АЦП рівний двом (біти CK1=1 CK0=1), то це складе 34-40 тактів мікроконтролера. Взнати точний час завершення аналогово-цифрового перетворення можна опитуючи біт ADCI – вкінці циклу перетворення контролер його встановлює.
Результат перетворення знаходитиметься в регістрах ADCDATAH:ADCDATAL. Для очищення інформації про канал, яку містить ADCDATAH, скористаємось оператором &.
Наступний фрагмент коду запустить аналогово-цифрове перетворення і збереже результат в змінні rslt:

```
unsigned int rslt;
ADCCON2 = 0x17;//Запуск АЦП
while (ADCI); //Очікування завершення циклу АЦП
rslt = ((ADCDATAH&0x0F)<<8) | ADCDATAL; //Збереження результату
```

В навчальному стенді до каналів АЦП приєднані такі пристрої

Таблиця 3 – Канали АЦП навчального стенду

Номер каналу	Порт	Пристрій

0	P1.0/ADC0	Інструментальний підсилювач AD627. Роз'єм X1
1	P1.1/ADC1	Диференціальний підсилювач. Роз'єм X2
2	P1.2/ADC2	Неінвертуючий підсилювач із подільником. Роз'єм X3
3	P1.3/ADC3	Інвертуючий підсилювач із зміщенням. Роз'єм X4
4	P1.4/ADC4	Неінвертуючий підсилювач із зміщенням та масштабуваннм. Роз'єм X5
5	P1.5/ADC5	Генератор імпульсів на NE555
6	P1.6/ADC6	Потенціометр маніпулятора «вліво-вправо»
7	P1.7/ADC7	Потенціометр маніпулятора «вверх-вниз»

b. Практична частина

Перший рівень

1. Створіть копію папки проекту. Назвіть її lab6. Відкрийте проект і доповніть його такими функціями:

`void InitADC()`	Вмикає АЦП та налаштовує параметри
`unsigned int ADCget(unsigned char channel)`	Запускає одиничне аналогово-цифрове перетворення з каналу channel, чекає його завершення та повертає результат
`unsigned int ADCget(unsigned char channel; unsigned char avg)`	Усереднює результати аналогово-цифрового перетворення з каналу channel по avg вибірках

2. Функцію `main()` змініть так, щоб після старту програми виконувались функції ініціалізації АЦП та очистки регістрів а основний цикл опитував сьомий канал АЦП, усереднюючи сигнал по 16 вибірках. Результати вимірювання відображайте на статичному індикаторі.
3. Відкомпілюйте та завантажте програму. Перевірте правильність її роботи.

Другий рівень

4. Змініть основний цикл так, щоб натискання клавіші 0..9 вмикало відповідний канал.

Контрольні питання

1. Для чого призначений пристрій вибірки/зберігання?
2. На яких принципах будують АЦП?
3. На якому принципі побудоване внутрішнє АЦП мікроконтролера ADuC841?
4. Нарисуйте типову схему спряження АЦП із зовнішніми вимірювальними колами.
5. Від чого залежить час аналогово-цифрового перетворення?
6. Що таке апертурний час? Якими міркуваннями слід керуватися встановлюючи його?
7. Як пов'язаний вихідний код з вхідною та опорною напругою АЦП ADuC841?
8. Визначте період АЦП, якщо тактова частота контролера 11.0592МГц, а байт ініціалізації ADCCON1 = 0xBC.

ЛАБОРАТОРНА РОБОТА № 6

Тема: Обробники переривань.
Новий матеріал: Система переривань контролера ADuC841, обробники переривань в С.

Короткі теоретичні відомості

Система переривань контролера ADuC841

Виконання основного потоку команд може перерватися для реагування на певні події. Стандартне ядро 8051 передбачає переривання по від'ємному фронті або нульовому рівні зовнішніх сигналів INT0 та INT1, переповненню таймерів T0, T1 завершенні отримання чи передачі байту послідовним інтерфейсом. ADuC841 розширює цей перелік перериванням таймера 2, АЦП перетворення, I^2C/SPI інтерфейсу, монітора живлення, годинника реального часу та сторожового таймера.
Опрацювання переривання відбувається так. Після того як відбулась певна подія (наприклад переповнився таймер, завершилося аналогово-цифрове перетворення чи отримання байту послідовним інтерфейсом) апаратна частина мікроконтролера встановлює біт-ознаку події (interrupt flag) (наприклад ADCI після завершення АЦП чи RI після одержання байту послідовним інтерфейсом). Якщо відповідне переривання дозволене і біт глобального дозволу переривань EA встановлений, то виконання основної програми переривається, адреса поточної команди зберігається в стеку, а керування передається інструкції за адресою, що визначається перериванням. Адреси переривань приведені в таблиці 5. Після опрацювання реакції на подію виконується інструкція **`reti`** яка дозволяє переривання і продовжує виконання програми із команди наступної після збереженої в стуці перед перериванням. Оскільки обробник переривання може бути викликаний в будь-який момент часу, то після свого завершення він повинен залишити вміст робочих регістрів контролера (ACC, B, PSW, DPTR, R0-R7) незмінним.
Як діятиме контролер якщо під час опрацювання переривання виникне нове? Ядро 8051 передбачає два рівні пріоритету переривання – низький та високий. Якщо вже відбувається опрацювання переривання низького пріоритету, то перервати його роботу зможе лише переривання з високим пріоритетом. Якщо під час відпрацювання переривання трапилося переривання такого ж пріоритету, то його відпрацювання відкладається до завершення роботи переривання яке відпрацьовується. Якщо одночасно очікують опрацювання декілька подій однакового пріоритету, то вони контролер обслуговуватиме їх в послідовності, приведеній в таблиці 4. Оскільки обробник призупиняє виконання основної програми, то для стабільної і швидкої роботи контролера слід реалізовувати обробники переривань що часто трапляються якомога швидшими.
Реакція на переривань ядра 8051 налаштовується за допомогою регістрів IE та IP. Контролер ADuC841 для цього додатково використовує регістр IEIP2.

IE – регістр дозволу переривань. Біти 0..6 відповідають за дозвіл відпрацювання певного переривання (таблиця 1). Встановлення сьомого біта рівним 0 забороняє будь-які переривання.

Таблиця 1 – Регістр дозволу переривань IE (адреса – 0xA8; побітова адресація дозволена)

Біт	Назва	Джерело	Ін.
7	EA	Глобальний дозвіл	0
6	EADC	Дозвіл переривання АЦП	0
5	ET2	Дозвіл переривання таймера 2	0
4	ES	Дозвіл переривання послідовного інтерфейсу	0
3	ET1	Дозвіл переривання таймера 1	0
2	EX1	Дозвіл зовнішнього переривання 1 (INT1)	0
1	ET0	Дозвіл переривання Таймера 0	0
0	EX0	Дозвіл зовнішнього переривання 0 (INT0)	0

Регістр IP задає пріоритет переривань. Його адреса 0xB8. Побітова адресація дозволена. Після старту регістр містить значення 0.

Таблиця 2 – Регістр пріоритету переривань IP (адреса:; побітова адресація;)

Біт	Назва	Джерело	Ін.
7	---	Зарезервовано	0
6	PADC	Високий пріоритет переривання АЦП	0
5	PT2	Високий пріоритет переривання таймера 2	0
4	PS	Високий пріоритет переривання послідовного інтерфейсу	0
3	PT1	Високий пріоритет переривання таймера 1	0
2	PX1	Високий пріоритет зовнішнього переривання 1 (INT1)	0
1	PT0	Високий пріоритет переривання Таймера 0	0
0	PX0	Високий пріоритет зовнішнього переривання 0 (INT0)	0

Регістр IEIP2 дозволяє та задає пріоритет деяких додаткових переривань. Його адреса 0xA9. Побітова адресація непередбачена. Після старту регістр містить значення 0xA0.

Таблиця 3 – Додатковий регістр дозволу і пріоритету переривань IEIP2

Біт	Назва	Джерело	Ін.
7	---	Зарезервовано	1
6	PTI	Високий пріоритет переривання лічильника часу	0
5	PPSM	Високий пріоритет переривання монітора живлення	1
4	PSI	Високий пріоритет переривання SPI/I^2C	0
3	---	Зарезервовано	0
2	ETI	Дозвіл переривання лічильника часу	0
1	EPSMI	Дозвіл переривання монітора живлення	0
0	ESI	Дозвіл переривання SPI/I^2C.	0

Таблиця 4 – Послідовність опрацювання переривань однакового пріоритету

Джерело	Пріоритет	Назва
PSMI	1(найвищий)	Монітор живлення
WDS	2	Сторожовий таймер
IE0	3	Зовнішнє переривання 0
ADCI	4	АЦП
TF0	5	Таймер/лічильник 0
IE1	6	Зовнішнє переривання 1
TF1	7	Таймер/лічильник 1
ISPI/I2CI	8	Інтерфейс SPI/I^2C
RI+TI	9	Послідовний інтерфейс
TF2+EXF2	10	Таймер/лічильник 2
TII	11(найнижчий)	Годинник реального часу

Таблиця 5 – Адреси вектора переривань

Джерело	Адреса	Номер Keil	Назва
IE0	0x0003	0	Зовнішнє переривання 0
TF0	0x000B	1	Таймер/лічильник 0
IE1	0x0013	2	Зовнішнє переривання 1
TF1	0x001B	3	Таймер/лічильник 1
RI+TI	0x0023	4	Послідовний інтерфейс
TF2+EXF2	0x002B	5	Таймер/лічильник 2
ADCI	0x0033	6	АЦП
ISPI/I2CI	0x003B	7	Інтерфейс SPI/I^2C
PSMI	0x0043	8	Монітор живлення
--	0x004B	9	*Не використовується*
TII	0x0053	10	Годинник реального часу
WDS	0x005B	11	Сторожовий таймер

Реалізація обробників переривань компілятором Keil

Обробник переривання – функція яка виконується як реакція на певну подію. Для того, щоб вказати компілятору Cx51, що певна функція є обробником переривання потрібно після заголовку функції вказати ключове слово **`interrupt`** разом із номером переривання. Назва функції – довільна. Функція не повинна приймати чи повертати жодних параметрів. Функцію-обробник не можна викликати з іншої функції.

Наприклад обробник переривання таймера 0 описується таким чином:

```
void OnT0() interrupt 1
{
  Counter++;
}
```

Компілятор Cx51 підтримує 32 джерела переривань які нумеруються з нуля. Відповідність між перериваннями ADuC841 та номерами обробників Keil приведені в таблиці 5.

Дозвіл переривання та налаштування джерела залишається обов'язком програміста. Ключове слово **`interrupt`** впливає на код програми таким чином:

- Якщо потрібно, то вміст регістрів ACC, B, DPH, DPL, PSW зберігається в стеку на початку виконання функції.
- Вміст використаних регістрів зберігається якщо не було вказано ключового слова **`using`**.
- Перед виходом із функції відновлюється вміст збережених регістрів.
- Для повернення із функції замість **`ret`** використовується інструкція **`reti`**.
- У таблиці переривань автоматично генерується інструкція переходу на функцію-обробник.

Практична частина

Дослідження ефекту "брязкоту" контактів

При замиканні чи розмиканні механічних контактів відбувається перехідний процес, під час якого надійний контакт встановлюється лише після серії перемикань. Нашим завданням буде дослідити його тривалість та кількість перемикань.

1. Створіть копію папки проекту, створеного на попередній лабораторні роботі.
2. Додайте до тексту програми визначення глобальної змінної `Counter` типу **`unsigned int`**.
3. Додайте функцію-обробник переривання від зовнішнього джерела 0 (INT0). Реакцією на від'ємний фронт сигналу INT0 має бути збільшення змінної Counter на одиницю, встановлення регістрів TH0:TL0 в 0 та запуск таймера (TR0 =1).
4. Додайте функцію-обробник переривання від переповнення таймера 0. Обробник повинен встановити змінну Counter в 0 і зупинити таймер.
5. Змініть основну програму таким чином, щоб вона після старту встановлювала таймер 0 в перший режим (регістр TMOD біти M1:M0=01), вмикала спрацювання переривання INT0 по фронту сигналу (біт IT0=1) та дозволяла переривання по IE0 та TF0 (встановити біти EX0, ET0 та EA – глобальний дозвіл). В основному циклі відображайте на статичному індикаторі значення змінної Counter, якщо воно не рівне нулю.
6. Відкомпілюйте та завантажте програму. Перевірте правильність її роботи.

Додаткове завдання

7. Перепишіть програму так, щоб на статичному індикаторі відображався час перехідного процесу в циклах мікроконтролера.

Контрольні питання

1. Які переривання підтримує стандартне ядро 8051?
2. Назвіть додаткові переривання, що реалізує контролер ADuC841?
3. Яким чином відбувається опрацювання переривання?
4. Що відбудеться якщо під час обробки переривання виникає необхідність обробити інше?
5. Скільки є рівнів пріоритету переривань?
6. Які вимоги до підпрограми обробки переривань?
7. Як описати обробник переривання мовою С для компілятора Cx51?
8. Як ключове слово **`interrupt`** впливає на код, що генерує компілятор?
9. Знайдіть і виправте помилки в коді:

a.

```
void onINT0() interrupt;
{
}
```

b.

```
char INT0() interrupt 5
{
}
```

c.

```
void onINT0() interrupt 0
{
  reti;
}
void main()
{
  onINT0();
}
```

ЛАБОРАТОРНА РОБОТА № 7

Тема: Модулі. Використання асемблера.
Новий матеріал: .

Короткі теоретичні відомості

Використання файлів заголовків

При написанні програм корисно групувати функції за призначенням в декілька відносно незалежних модулів. Таким чином отримуємо набір блоків, які можна буде легко використати при реалізації наступних проектів.
Кожний такий блок прийнято розміщати в двох файлах.

- В .c файлі розміщують реалізацію модуля: описують тіла функцій та оголошують глобальні змінні. Тут також оголошують глобальні змінні, функції, макроси та типи, які є внутрішні по відношенню до модуля і мають бути "приховані" від його користувача.
- В .h файлі розміщують інформацію про спосіб користування модулем (інтерфейс модуля): оголошують прототипи функцій, макроси, описи типів та посилання на змінні які мають бути доступні для користувача модуля. В коментарях варто описати призначення і спосіб використання модуля, його функцій та доступних змінних.

Для прикладу розглянемо модуль UART. Файл uart.h містить декларації прототипів функцій, що мають бути доступні користувачу (функції ініціалізації, отримання та відправки байт, очистки вхідного буфера, визначення розміру вхідного і вихідного буферу). З цим файлом найперше буде працювати користувач модуля. У файлі uart.c розміщені описи змінних та функцій, які безпосередньо відповідають за реалізацію модуля. Таким чином ми відділяємо і приховуємо реалізацію, що полегшує користування модулем.
Файл uart.h

```
/*********************************************************************
* uart.h                                                             *
* Модуль обміну асинхронним інтерфейсом. Файл заголовків.            *
*********************************************************************/
#define RX_BUFFER_SIZE 20      /*Розмір вхідного буфера*/
#define TX_BUFFER_SIZE 20      /*Розмір вихідного буфера*/
void UARTinit();               //Ініціалізація UART
bit SetBaud(unsigned char BR); //Встановлює частоту передачі
void putchar(char c);          //Кладе байт у вихідний буфер.
                               //Якщо він переповнений,
                               //то чекає звільнення
char getchar(void);            //Виймає байт із вхідного буфера.
                               //Якщо буфер пустий, то очікує приходу.
unsigned char RxCount();       //Кількість байт у вхідному буфері
unsigned char TxCount();       //Кількість байт у вихідному буфері
void FlushRxBuffer();          //Очистка вхідного буферу
extern bit rx_buffer_overflow; //Ознака переповнення вхідного буферу
```

Файл uart.c (скорочений)

```
/*********************************************************************
* uart.c                                                             *
* Модуль обміну асинхронним інтерфейсом. Реалізація.                 *
*********************************************************************/
#include "uart.h"
```

```
  ...


//Опис змінних модуля
bit  TxEnableB;                          //Ознака активності передачі
char idata rx_buffer[RX_BUFFER_SIZE]; //Вхідий буфер
unsigned char rx_wr_index=0, rx_rd_index=0, rx_counter=0;
...

//Визначення функцій
void usart_rx_isr(void) interrupt 4
{
// USART0 Receiver interrupt service routine
  ...
}

char getchar(void)
{
//Виймає байт із вхідного буфера. Якщо буфер пустий, то очікує байт.
  ...
  return ...;
}
...
```

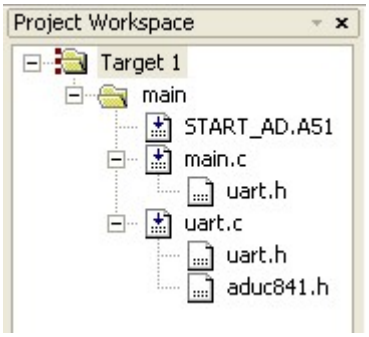


Рисунок 1 – Дерево проекту з підключеним модулем

Для того, щоб скористатися модулем слід додати .c файл до проекту (рисунок 1), а .h файл підключити до програми за допомогою директиви `#include`, що наказує компілятору вставити текст .h файлу в основний файл. Таким чином оголошення .h файлу стають доступні для основної програми на етапі компіляції.

Наприклад, наступна програма використовує модуль uart, щоб організувати функцію луни (передає кожний прийнятий байт).

```
#include "uart.h"

void main()
{
  UARTinit();
  while (1)
  {
    putchar(getchar());
  }
}
```

Компіляція проекту в середовищі Keil

Перетворення вхідних текстів програми на вихідний .hex файл відбувається в декілька етапів.

Компіляція. Тексти написані мовою assembler обробляються компілятором A51.EXE, тексти написані мовою C – компілятором C51.EXE. Якщо в тексті програми компілятор знаходить посилання на інший файл (директива #include),

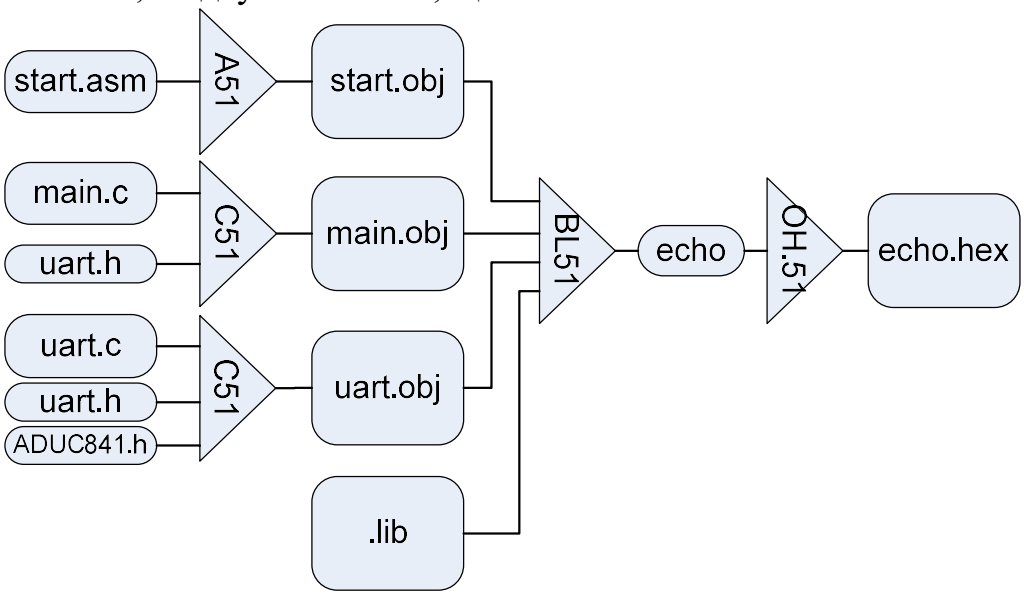
то він інтерпретує його, як частину компільованого файлу. Результатом роботи компілятора є **об'єктний файл (.obj)** та файли лістингу (.lst, .i). Файли лістингу містять вихідний текст програми з повідомленнями про помилки компіляції та службовою інформацією. Об'єктний файл містить:

- код (команди мікроконтролера);
- інформацію про глобальні змінні;
- назви символів (змінних та функцій), що імпортуються модулем (external);
- назви символів (змінних та функцій), що експортуються модулем (public);
- необхідні файли бібліотек;
- інформацію для відладки.

Більшість коду, розміщеного в .об'єктному файлі є відносним, тобто він може бути розміщений будь-де в програмі. Абсолютні адреси змінних та функцій теж невідомі. Замість адреси символу компілятор вставляє назву. Реальний розподіл символів в пам'яті буде виконаний на наступному етапі.

Компонування. Щоб отримати виконавчий файл потрібно розмістити код та змінні по реальних адресах в просторі програм і даних. Для цього лінкер (linker) L51.EXE обробляє набір об'єктних файлів, які були отримані на етапі компіляції разом із бібліотеками функцій, які реалізують такі стандартні функції мови С як додавання чи ділення багатобайтних чисел.

На етапі компановки програми компонувальник складає окремі фрагменти коду та констант по конкретних фізичних адресах, одночасно замінюючи символьні посилання на фізичні адреси програми. Компонувальник також відповідальний за розміщення змінних в пам'яті даних і складання дерева викликів процедур. Оскільки в мікроконтролері MCS51 звертання до змінних, розміщених в стеці ускладнене, виділенням пам'яті під локальні змінні займається компонувальник, слідкуючи за тим, щоб змінні із



Практична частина

Термінал

1. Напишіть програму, що приймає прості команди в текстовому вигляді по УАПП та відповідає на них. Реалізуйте команди відображення рядка на символьний рідкокристалічний дисплей, семисигментний індикатор та матрицю світлодіодів. Також передбачте команду перезавантаження стенда.
2. Відкомпілюйте та завантажте програму. Перевірте правильність її роботи.

Контрольні питання

1. З яких етапів складається перетворення вихідного коду в готову до виконання програму?
2. В чому полягає різниця між компіляцією та компонуванням?
3. Назвіть та коротко опишіть призначення компонентів системи Keil.
4. Що таке бібліотека?

ПРИКЛАДИ ПРОГРАМ ДО ЛАБОРАТОРНИХ РОБІТ

Лабораторна робота №1. Використання операторів та функції в програмах написаних мовою С.

Приклад відображення числа 5_{10} (00001001_2) на лінійці світлодіодів. Вміст файлу main.c

```
#include "ADUC841.h"

void write(unsigned char Addr, unsigned char Dat)
{
  WR = 1;
  P0 = Dat;
  P2 &= 0xF0;
  P2 |= (Addr&0x0F);
  P2 &= 0xF0;
}


void main()
{
  write(7, ~5);
  while (1)
  {
  }
}
```

Лабораторна робота №2. Змінні, константи та масиви.

Приклад відображення числа 1234_{16} на семисегментному індикаторі. Вміст файлу main.c

```
#include "ADUC841.h"

void write(unsigned char Addr, unsigned char Dat)
{
  WR = 1;
  P0 = Dat;
  P2 &= 0xF0;
  P2 |= (Addr&0x0F);
  P2 &= 0xF0;
}

const unsigned char code Hexto7[16] =
 {0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
  0x80, 0x98, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E};

void Static(unsigned int A)
{
  write(1, Hexto7[(A    )&0x0F]);
  write(2, Hexto7[(A>>4 )&0x0F]);
  write(3, Hexto7[(A>>8 )&0x0F]);
  write(4, Hexto7[(A>>12)&0x0F]);
}

void main()
{
  Static(0x1234);
  while (1)
  {
  }
}
```

6. РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

1. Kernighan B. W. The C Programming Language. Second Edition [Text] / Brian Kernighan, Dennis M. Ritchie – New Jersey: Prentice Hall, 1988. – 272 p.
2. Керниган Б. Язык С [Текст] / Керниган Б., Деннис Ричи;

Допоміжна

3. ADuC841 AnalogDevices MicroConverter® 12-Bit ADCs and DACs with Embedded High Speed 62-kB Flash MCU [Electronic resource]

Інформаційні ресурси

4. http://www.analog.com/