

О. В. Бєлявський, О. В. Косенко, А. О. Бреус

**ОСНОВИ МІКРОПРОЦЕСОРНОЇ ТЕХНІКИ
ТА ЇЇ ПРОГРАМУВАННЯ МОВОЮ С**

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет
«Харківський авіаційний інститут»

О. В. Бєлявський, О. В. Косенко, А. О. Бреус

**ОСНОВИ МІКРОПРОЦЕСОРНОЇ ТЕХНІКИ
ТА ЇЇ ПРОГРАМУВАННЯ МОВОЮ С**

Навчальний посібник до практичних робіт

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет
«Харківський авіаційний інститут»

О. В. Бєлявський, О. В. Косенко, А. О. Бреус

**ОСНОВИ МІКРОПРОЦЕСОРНОЇ ТЕХНІКИ
ТА ЇЇ ПРОГРАМУВАННЯ МОВОЮ С**

Навчальний посібник до практичних робіт

УДК 621.865.8:004.312.466(076.5)
Б44

Рецензенти: д-р техн. наук, проф. С. С. Добротворський,
д-р техн. наук, проф. В. Ф. Сорокін

Бєлявський, О. В.

Б44 Основи мікропроцесорної техніки та її програмування мовою С [Електронний ресурс] : навч. посіб. до практ. робіт / О. В. Бєлявський, О. В. Косенко, А. О. Бреус. – Харків : Нац. аерокосм. ун-т «Харків. авіац. ін-т», 2025. – 114 с.

ISBN 978-966-996-056-6

Викладено елементи базового курсу лекцій з дисципліни. Розглянуто методи програмування алгоритмічною мовою С і методи її використання під час програмування мікроконтролерів родини AVR у середовищі AVRSTUDIO 5, а також методи створення програмного забезпечення для типових вузлів систем автоматики. Запропоновано приклади програм, що є цілком відлагодженими й придатними для вирішення практичних завдань навчального курсу. Наведено таблиці команд процесорів родини AVR, що дає читачу змогу використовувати мову асемблера мікропроцесорів родини AVR під час програмування.

Для здобувачів вищої освіти технічних ЗВО, але також може бути корисним для аспірантів та інженерно-технічних працівників, що вирішують питання автоматизації технологічних процесів.

Іл. 28. Табл. 12. Бібліогр.: 7 назв

УДК 621.865.8:004.312.466(076.5)

© Бєлявський О. В., Косенко О. В.,
Бреус А. О., 2025

© Національний аерокосмічний університет
«Харківський авіаційний інститут», 2025

ISBN 978-966-996-056-6

ВСТУП

Мікропроцесори (МП) і МП-техніка давно й міцно увійшли в наше життя, полегшили, а в багатьох випадках і докорінно змінили участь людини у виробничих процесах. Починаючи вивчення мікропроцесорних пристроїв автоматики, слід визначити коло джерел, які є базою цього курсу. По-перше, це булева алгебра – основа розуміння процесів, які відбуваються у складових вузлах МП; по-друге, цифрова схемотехніка, що дає змогу оцінити порядок, методику й час вирішення завдання, оптимізувати структуру й методи оброблення даних; по-третє, апаратна структура ресурсів МП пристрою та методів програмного доступу до них у вигляді прикладів програмування, наприклад, мовами С або асемблера; по-четверте, інтегроване середовище розроблення програмного забезпечення з редактором, компілятором, налагоджувачем, симулятором і програматором, який зазвичай має бути виготовлено (або придбано), а також сам кристал МП пристрою зі всіма потрібними зовнішніми елементами.

На відміну від багатьох методичних рекомендацій сучасності у запропонованому виданні починається розгляд теми з вивчення цифрової схемотехніки й методів синтезу цифрових пристроїв автоматики як базових складових елементів електронної техніки взагалі й мікропроцесорної зокрема. Такий підхід, на думку авторів, дасть можливість здобувачу освіти охопити більшу частину комплексу проблем, які постають перед інженером-проектувальником МП-пристроїв, і впевнено вирішувати найважливіші завдання.

Такий підхід базується на переконанні авторів у необхідності глибокого оволодіння булевою алгеброю як «найкрасивішою» дисципліною сучасності всіх, хто починає вивчати основи мікропроцесорної техніки й автоматики за фахом G01 «Робототехніка». Таке вивчення стане в нагоді при вивченні курсів «Гідроелектромеханічні перетворювачі енергії», «Пневмогідроавтоматика» та інших.

Історія створення обчислювальних систем починається 1946 року в США, де на замовлення ВМС було створено обчислювальну машину «МАРК-1», яка потребувала енергії цілої електростанції, мала розрядність даних 16 бітів і працювала на цифрових елементах релейної техніки, проте вона забезпечувала відстеження радянських підводних човнів в акваторії Всесвітнього океану. Швидкодія першої машини не перевищувала 400 Гц – максимальної швидкодії релейного елемента. Сучасні ЕОМ побудовано на базі мікроінтегральної елементної бази, де розмір активного елемента становить 13 нм, швидкодія наближається до десятків гігагерців, а місткість накопичувача даних становить кілька терабайтів.

У навчальному посібнику розкрито методи програмування найпоширенішого класу мікропроцесорних пристроїв світового лідера

продажів однокристальних мікроЕОМ – фірми ATMEЛ – і надається змога долучитися до цього захопливого процесу.

З огляду на дидактичну організацію навчальний матеріал подано за планом, у якому передбачено викладення на початку теоретичної частини знань з наступним їх закріпленням у практичних завданнях, наведених у кожному розділі.

Схемотехніка цифрових електронних пристроїв – це інженерне втілення принципів булевої алгебри в практичну реалізацію електронної схеми, яка має виконувати конкретні функції генерування, перетворення й зберігання сигналів.

Здобувачам освіти, які вперше стикаються з цифровою й мікропроцесорною технікою, автори рекомендують насамперед освоїти підручник П. Хоровіца, У. Хіла [1], розрахований саме на них. Для більш повного розуміння курсу після початкового ознайомлення з предметом можна долучитися до вивчення фундаментальної праці У. Тітце та К. Шенка [2], яка містить базові відомості з питань принципу дії та застосування напівпровідникових елементів узагалі і мікропроцесорних пристроїв зокрема. Для вивчення методів програмування мовою С можна порадити популярне видання Уллі Соммера [3], яке надає знань з питань програмування мовою С скейтч *ARDUINO* МП пристроїв платформи *ARDUINO/FREEDUIN*, і популярне видання [4], у якому розглядається питання програмування мовою С для *AVR* і *PIC* мікроконтролерів. Видання містить велику кількість повністю готових до використання програм, довідник функцій і макровизначень системи програмування *WINAVR*.

1. ЕЛЕМЕНТИ ТЕОРІЇ ДИСКРЕТНИХ АВТОМАТІВ

1.1. Логічні функції та методи їх подання

Логічною функцією (ЛФ) називають величину, яку можна подати у дискретному вигляді скінченною кількістю дискретних значень (логічних рівнів). Найпростіший випадок – двійкова логіка: кліщі робота затиснуто/розтиснено, механізм увімкнено/вимкнено, піч нагрівається/охолоджується тощо. Основа логіки – два стани логічної змінної: істина – 1 і хибність – 0. Для трійкової логіки таких станів буде три: +1, 0, –1 (наприклад, +1 – це набір висоти, –1 – зниження, 0 – горизонтальний політ) тощо.

У цьому розділі розглядається проектування двійкових механізмів автоматики, які найчастіше використовуються в промислових системах. Спочатку логічні блоки систем керування будувалися на релейно-контактних елементах, і тому мова релейно-контактних схем є природною мовою опису таких пристроїв.

У табл. 1.1 наведено релейно-контактні схеми основних логічних функцій, які можна подати і в аналітичному, і в табличному вигляді. Кожна логічна змінна має індивідуальне буквенно-цифрове ім'я, яке належить до логічних висловів. Крім зазначених методів ЛФ можна задати графом бінарного алгоритму, програмною реалізацією деякої алгоритмічної мови, функціональною або принциповою електричною схемою, часовою діаграмою функціонування. На рис. 1.1 показано приклад подання логічного пристрою складання за модулем 2 (без переповнення).

Основні положення двійкової алгебри

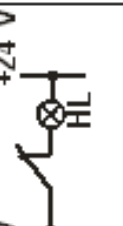
Математичний апарат подання цифрових пристроїв базується на алгебрі англійського математика Дж. Буля (1815–1864), де логічні зв'язки визначаються відношенням еквівалентності («=»), і трьома логічними операціями: диз'юнкції, що позначається як «+» або «V», кон'юнкції– «•» або «^» і заперечення (верхня риска) або «!» (для запису мовою С).

При виконанні всіх трьох операцій відношення еквівалентності визначаються такими виразами:

$$\begin{array}{lll} 0+0=0; & 0\cdot 0=0; & \bar{0}=1; \\ 1+0=1; & 1\cdot 0=0; & \bar{1}=0. \\ 0+1=1; & 0\cdot 1=0; & \\ 1+1=1; & 1\cdot 1=1; & \end{array}$$

Ці постулати дають змогу обґрунтувати закони двійкової алгебри.

Таблиця 1.1

Аналітичне подання ЛФ						
A	B	F	$F = \overline{AB}$	$F = A + B$	$F = \overline{A+B}$	$F = \overline{A}$
Табличне подання ЛФ						
0	0	0	1	0	1	1
0	1	0	1	1	0	1
1	0	0	1	1	0	0
1	1	1	0	1	0	0
Графічне подання ЛФ						
Тип елемента	2I	2I-НІ	2АБО	2АБО-НІ	Виключне АБО	НІ
УГО ЛЕ 						
Релейно-контактне подання ЛФ						
						

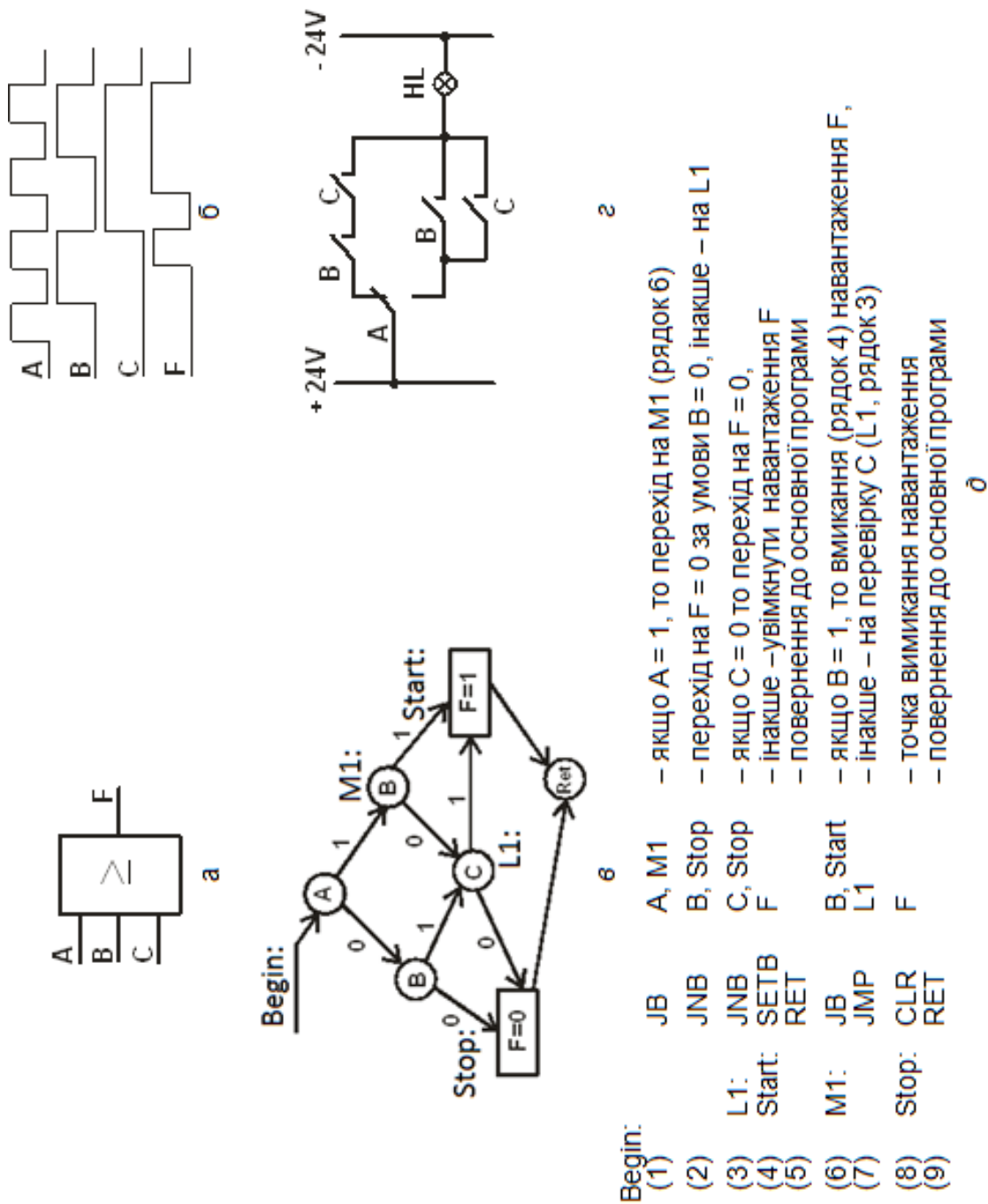


Рис. 1.1. Методи задання ЛФ мажоритарного елемента 2 із 3: а – функціональне УГО; б – часова діаграма; в – граф бінарного алгоритму; г – релейно-контактна схема; д – запис алгоритму мовою асемблера EOM Intel 8051

Призначення елемента – вибір: якщо два сигнали із трьох є істинними, то на виході елемента буде високий логічний потенціал, інакше – низький. Застосування: прийняття рішень, наприклад, у телеметричних каналах зв'язку.

Графічне подання алгоритму – обов'язковий крок на шляху складання програмного забезпечення деякою алгоритмічною мовою програмування. Для авторів – це мова асемблера процесора *Intel 8051*, ознайомлення з яким рекомендуємо почати з читання посібника [7]. Кожна вершина графа (рис. 1.1, в) відповідає умовно-логічному оператору мови *Jump if Bit set (JB)* або *Jump if Not Bit set (JNB)*, які аналізують або рівність, або нерівність деякої логічної змінної значенню «істина». Ім'я логічної змінної поміщено у вершині графа. Команда *JMP* символізує безумовний перехід за програмою на задану мітку. Змінну записують у другому параметрі команди, а мітку, на яку слід перейти в разі виконання умови, – у третьому.

Команда *CLR* викликає очищення певного біта (вимкнення навантаження), а команда *SETB* – його встановлення в одиницю (увімкнення). Команда *RET* – це наказ повернутися в тіло головного модуля програми, звідки й було викликано цю підпрограму. Мітки в мові Асемблера (як і в інших мовах) виділяються двокрапкою та записуються в крайній лівій позиції рядка, оператор записується в наступних восьми позиціях; за оператором іде операнд (або два: один – джерело, інший – приймач); ознака «;» відокремлює від програми коментар розробника, який на програму ніяк не вплине, а в Асемблері є просто необхідним. У дужках розміщують номери рядків, які компілятор розставить автоматично в лістингу трансляції програми. Відразу зазначимо момент, що погано сприймається здобувачами освіти: до рядка 2 програма перейде, а то й буде виконувати умову в рядку 1, тобто якщо змінна $A = 0$. Такого самого правила дотримуються для всіх операторів *JB*, *JNB*.

1.2. Аксиоми й постулати булевої алгебри

Закони одинарних елементів:

– універсальної множини:

$$x + 1 = 1;$$

$$x \cdot 1 = x;$$

– нульової множини:

$$x + 0 = x;$$

$$x \cdot 0 = 0.$$

Закони заперечення (теорема Моргана):

– подвійного заперечення:

$$\overline{\overline{x}} = x;$$

– додатковості:

$$x + \bar{x} = 1;$$

$$x\bar{x} = 0;$$

– двоїстості:

$$\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2;$$

$$\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2.$$

Закони комбінаційні:

– тавтології:

$$x + x = x;$$

$$\bar{\bar{x}} = x;$$

– комутативні, або переміщувальні:

$$\bar{x}_1 + \bar{x}_2 = \bar{x}_2 + \bar{x}_1;$$

$$\bar{x}_1 \bar{x}_2 = \bar{x}_2 \bar{x}_1;$$

– асоціативні, або погоджувальні:

$$(\bar{x}_1 + \bar{x}_2) + \bar{x}_3 = \bar{x}_1 + (\bar{x}_2 + \bar{x}_3);$$

$$(\bar{x}_1 + \bar{x}_2) \bar{x}_3 = \bar{x}_1 (\bar{x}_2 + \bar{x}_3);$$

– дистрибутивні, або розподільчі:

$$\bar{x}_1 (\bar{x}_2 + \bar{x}_3) = \bar{x}_1 \bar{x}_2 + \bar{x}_1 \bar{x}_3;$$

$$\bar{x}_1 + (\bar{x}_2 \bar{x}_3) = (\bar{x}_1 + \bar{x}_2) (\bar{x}_1 + \bar{x}_3);$$

– абсорбції, або поглинання:

$$\bar{x}_1 + \bar{x}_2 \bar{x}_1 = \bar{x}_1;$$

$$\bar{x}_1 (\bar{x}_2 + \bar{x}_1) = \bar{x}_1;$$

– склеювання:

$$!x_1 !x_2 + x_1 !x_2 = !x_2;$$

$$(x_1 + x_2)(x_1 + !x_2) = x_1.$$

Двійкова алгебра є дуальною щодо складання й множення. Цю властивість двоїстості алгебри логіки узагальнив Шеннон у такій теоремі: інверсія довільної комбінації двійкових змінних, пов'язаних знаками $A * B$ та $A + B$, є еквівалентною заміні в ній значень змінних їх інверсіями при одночасному зміні знака логічної операції $!(x_1 + x_2)$ на $!x_1 \cdot !x_2$ і навпаки, наприклад:

$$!(x_1 x_2 + x_3 x_4) = (!x_1 + !x_2)(!x_3 + !x_4).$$

Перемикальною функцією (ЛФ) називають змінну, значення якої залежить від значень інших двійкових змінних – аргументів. Перемикальна функція задається як відповідність її значення кожному набору (комбінації) значень її аргументів. Перемикальні функції вважаються різними, якщо відрізняються значеннями хоча б одного набору.

При n аргументах кількістю наборів $N = 2^n$, а кількість перемикальних функцій $M = 2^N = 2^{2^n}$.

Запис наборів ЛФ легко виконується у вигляді натурального ряду двійкових чисел. Набори нумеруються за значеннями двійкових чисел від 0 до $2^n - 1$.

Суперпозиція ЛФ полягає в тому, що її аргументи самі можуть бути ЛФ, тобто якщо

$$y = f(z_1, z_2) \text{ і } z_1 = f_1(x_1, x_2), z_2 = f_2(x_3, x_4),$$

то

$$y = f_1(x_1, x_2, x_3, x_4).$$

Використовуючи цю властивість, можна збільшувати кількість аргументів і нарощувати ПФ від найпростішої до скільки завгодно складної. Мінімальний набір таких елементарних функцій, який з допомогою суперпозиції може створити будь-яку ПФ, називають функціонально повним, або базисом. Це базиси 2І–НІ $\overline{x_1 x_2}$ – «штрих Шеффера», і 2АБО–НІ $\overline{x_1 + x_2}$ – «стрілка Пірса». Ці набори становлять основу цифрової електроніки. Використовуючи закони двійкової алгебри, можна здійснювати однозначний перехід із одного базису в інший.

Набір І, АБО, НІ називають основним функціонально повним. Стосовно нього розроблено сучасні методи синтезу.

Під дискретним пристроєм розуміють керувальний пристрій, який перероблює апріорну й поточну інформацію на керувальну, причому носіями всіх перелічених складових інформації є дискретні за рівнем і в часі сигнали. Це означає, що стан сигналу кожного входу (виходу) автоматизованого пристрою характеризується двома рівнями: мінімальним, що умовно позначається нулем, і максимальним, який позначається одиницею. За наявності декількох входів (виходів) комбінацію мінімальних і максимальних (нульових та одиничних) рівнів можна сприймати двояко:

- як наявність або відсутність на кожному вході (виході) задавального або командного сигналу, що надходить на будь-який виконавчий елемент;

- як кодову комбінацію, що позначає, наприклад, двійкове число, кількість розрядів n якого дорівнює кількості входів (виходів). Отже, можна казати про багаторівневий сигнал із кількістю рівнів $K = 2^n$.

У першому випадку дискретний логічний автоматизований пристрій вирішує завдання логічного аналізу ситуації, що визначається деякими факторами, що діють або яких немає, у другому – арифметичний пристрій, що здійснює перетворення інформації шляхом виконання арифметичних операцій над числами.

Як логічні, так і арифметичні пристрої реалізуються шляхом застосування обмеженої кількості типових елементарних операцій над дискретними сигналами й подібних методів синтезу й аналізу.

Дискретизація в часі означає, що протягом скінченного інтервалу часу, який називають тактом, стан сигналу або комбінації сигналів залишається постійним. Занумерована послідовність таких тактів утворює автоматний час.

Дискретні керувальні пристрої відповідно до вимог технологічного циклу поділяють на комбінаційні й послідовні. Для перших характерним є те, що комбінація станів вихідних сигналів Y_{ki} у певному такті t_i точно визначається комбінацією вхідних сигналів X_i у цьому такті часу, тобто

$$Y_{ki} = \lambda(X_i),$$

де λ – оператор (алгоритм) перетворення.

Комбінаційні автомати – це логічні пристрої, які не мають пам'яті.

У послідовних автоматах комбінація станів виходів у такті t_i залежить не тільки від комбінації станів входів у цьому такті, а й від того, якими були комбінації станів входів у n попередніх тактах, тобто

$$Y_{ni} = \lambda(X_i; X_{i-1}; \dots; X_{(i-n)}).$$

Їх називають також автоматами з ненульовою пам'яттю.

За характером реакції на вхідний вплив розрізняють *асинхронні автомати*, у яких змінення стану виходів відбувається одразу після змінення сигналів входу, і *синхронні*, у яких змінення стану виходів відбувається лише в тому випадку, якщо після змінення вхідних сигналів надійде сигнал, який синхронізує або дає дозвіл на перетворення.

Для створення найскладнішого дискретного автомата будь-якого типу досить мати елементи, що реалізують операцію АБО-НІ (інверсія диз'юнкції, або операція Пірса) або операцію І-НІ (інверсія кон'юнкції, або операція Шеффера), тобто так званий функціонально повний набір.

Послідовні автомати відрізняються від комбінаційних також наявністю зворотних зв'язків, за якими на входи автомата подаються сигнали, що відповідають стану виходів у попередньому такті.

Електромагнітне реле (рис. 1.2) є основою автоматики диспетчерських пунктів керування залізничним транспортом. На токарному верстаті з ЧПК 16K20T1Ф3 таких пристроїв встановлено понад 100 одиниць.



Рис. 1.2. Електромагнітне реле типу РКМП (6 груп, 24 В, паспорт РС 4.523.621, Ag – 0.71г)

1.3. Синтез дискретного автомата

Алгоритм синтезу дискретного автомата являє собою:

1) складання завдання на розроблення як змістовного опису, де формулюються вимоги з боку об'єкта керування до керувального автомата, тобто вимоги до поведінки виконавчих пристроїв об'єкта: на який виконавчий елемент об'єкта і в якому такті має подаватися мінімальний (0) або максимальний (1) рівень сигналу певного виходу автомата, щоб з урахуванням стану задаваних виходів і входів, які сприймають сигнали зворотних зв'язків, забезпечити необхідне функціонування керованого об'єкта;

2) одержання формальної моделі автомата на основі змістовного опису у вигляді рівнянь, таблиць, графів, матриць тощо;

3) отримання на основі моделі об'єкта вихідної (мінімальної) форми аналітичного подання оператора перетворення, що реалізується автоматом і забезпечує необхідний закон функціонування керованого об'єкта. Ця частина синтезу є основною і базується на аксіомах і постулатах булевої (двійкової) алгебри;

4) розроблення принципової схеми автомата на основі отриманої в п. 3 його формальної моделі й використання певної елементної бази;

5) розроблення й узгодження вузлів зв'язку автомата з оператором і керованим об'єктом, захист від перешкод тощо.

1.4. Синтез комбінаційних логічних схем

Оператори, що реалізуються комбінаційними автоматами, встановлюють однозначний зв'язок комбінацій вхідних і вихідних змінних для кожного значення автоматного часу t_i :

$$y_{1i} = \lambda(x_{1i}x_{2i}\dots x_{ni});$$

.....

$$y_{mi} = \lambda_m(x_{1i}x_{2i}\dots x_{ni}).$$

Для створення складних комбінаційних автоматів використовують елементарні комбінаційні логічні елементи, якими реалізується функціонально повний набір двійкових функцій. Елементарним комбінаційним називатимемо автомат, що реалізує ЛФ двох змінних.

Синтез комбінаційних автоматів виконується при статичній частині такту, тобто інерційність логічних елементів не враховується. Синтез комбінаційних автоматів починається від змістовного опису функціонування об'єкта й складання таблиці істинності автомата.

Для отримання аналітичного виразу, що визначає необхідну структуру автомата, записуються вихідні форми ЛФ у вигляді досконалої диз'юнктивної (ДДНФ) або кон'юнктивної (ДКНФ) нормальної форми.

Диз'юнктивна нормальна форма є диз'юнкцією (сумою) мінтермів, а кон'юнктивна – кон'юнкцією (добутком) макстермів.

Мінтермом, або конституентною одиницею, називають логічну функцію, що набуває значення одиниці тільки при одному наборі змінних. Утворюється як кон'юнкція всіх змінних вхідних з інверсією тих, які в певному наборі дорівнюють нулю. Кількість мінтермів дорівнює кількості наборів.

Макстермом, або конституентною нуллю, називають логічну функцію, що набуває значення нулля тільки при одному наборі змінних. Утворюється як диз'юнкція вхідних змінних, де змінні набору, що дорівнює одиниці, взято з інверсією. Кількість макстермів дорівнює кількості наборів.

Ці дві форми є еквівалентними. При мінімізації зручно користуватися ДДНФ, але є підручники з формальної логіки, у яких описується лише ДКНФ.

Розглянемо методи синтезу автоматів з мінімальною кількістю елементів, які мають назву мінімальних, або оптимальних.

Метод Квайна

Цей метод складається з таких етапів.

1. За таблицю істинності (табл. 1.2) записують ДДНФ. Кількість мінтермів дорівнює кількості наборів, де функція, що синтезується, дорівнює одиниці, а кількість співмножників у кожному наборі – повній кількості аргументів. Над аргументами, що набувають у цьому наборі значення нуля, ставиться знак інверсії:

$$Y = x_1 \overline{x_2} + \overline{x_1} x_2.$$

Таблиця 1.2

x_1	0	1	0	1
x_2	0	0	1	1
y	0	1	1	0

2. В отриманій ДДНФ знаходяться суміжні мінтерми (відрізняються знаком лише одного аргументу). Склеюючись за цим аргументом, суміжні мінтерми утворюють імпліканти – кон'юнкції з кількістю співмножників, меншою на одиницю. Імпліканти, своєю чергою, можуть склеюватися між собою. Імпліканти, що не склеюються, називають простими. Суміжні мінтерми й імпліканти мають такий вигляд:

Суміжні мінтерми	Суміжні імпліканти	Проста імпліканта
$\left. \begin{array}{l} x_1 x_2 \overline{x_3} \overline{x_4}, \\ x_1 x_2 \overline{x_3} x_4, \end{array} \right\}$	$\left. \begin{array}{l} x_1 x_2 \overline{x_3}, \\ \overline{x_1} x_2 \overline{x_3}, \end{array} \right\}$	$x_2 \overline{x_3}.$
$\left. \begin{array}{l} \overline{x_1} x_2 \overline{x_3} \overline{x_4}, \\ \overline{x_1} x_2 \overline{x_3} x_4, \end{array} \right\}$	$\left. \begin{array}{l} \overline{x_1} x_2 \overline{x_3}, \\ \overline{x_1} x_2 x_3, \end{array} \right\}$	

3. Отримана скорочена ДДНФ зводиться до тупикової форми шляхом:

а) усунення надлишкових імплікант (надлишковою називають імпліканту, що «покриває» мінтерми, уже покриті іншими імплікантами);

б) застосування закону поглинання для простих імплікант і решти мінтермів

$$\overline{x_1} \overline{x_2} + x_1 \overline{x_2} \overline{x_3} \overline{x_4} = \overline{x_1} \overline{x_2};$$

в) проведення випробування окремих членів. Для цього відкидають один із імплікантів, а у вирази, які залишилися, підставляють значення аргументів, що перетворюють випробовуваний член на одиницю. Якщо частина, що залишилася при цьому, тотожно дорівнює одиниці, то

випробовуваний член є зайвим, і навпаки. Метод Квайна застосовується до аргументів, кількість яких не перевищує шести. При більшій кількості застосовують машинний метод мінімізації Мак-Класки.

Метод Вейча – Карно

Найбільшого поширення сьогодні набув метод карт Карно. Суть його полягає в табличному поданні мінтермів ДДНФ і графічної інтерпретації їх склеювання. Карта Карно – це таблиця з кількістю клітин, що дорівнює кількості наборів аргументів. У кожену клітину можна вписати один набір (мінтерм), причому суміжні мінтерми мають бути в сусідніх клітинах. Це забезпечується спеціальним підбором мінтермів для кожного клітини. Щоб спростити це завдання, застосовують різні способи маркування сторін карток Карно, на рис. 1.3 показано два найбільш поширених.

На кожній зі сторін карти половина клітин відповідає одиничному значенню одного аргументу, а друга половина – нульовому. На протилежних гранях аналогічне маркування іншого аргументу робиться зі зсувом на одну клітину. У клітинках тих наборів, де функція набуває одиничного значення, пишеться 1, де нульового – 0.

При іншому способі аргументи розбиваються на дві групи з неоднаковою або однаковою кількістю аргументів. Для однієї групи X_1X_2 (перші номери) відводяться стовпці, для іншої X_3X_4 (останні номери) – рядки. Пряме або інверсне значення аргументу визначається двійковим цифровим маркуванням стовпців і рядків.

Охоплюючи одиничні клітини контурами, склеюємо суміжні мінтерми. Контур з двох клітин відповідає одноступінчасте склеювання, з чотирьох – двоступінчасте і т. д. Кількість клітин у контурі може бути лише парним. В утворюваних імплікантах (простих) залишаються змінні, які в контурі склеювання не змінювали знак. Їх кількість дорівнює $(n - k)$, де n – кількість аргументів.

Щоб функція була мінімальною, необхідно накрити всі одиниці мінімальною кількістю контурів максимальної площі. Контури можуть перетинатися. У тупиковій алгебричній формі, отриманій з карти Карно, це означає, що кількість контурів дорівнює кількості доданків. Від площі контуру залежить кількість співмножників у доданку; перетин контурів означає тавтологію мінтерму:

$$x_1x_2 + x_1x_2 = x_1x_2.$$

Розглянемо приклад мінімізації ЛФ чотирьох логічних змінних, заданої одиничним набором.

Приклад 1. Мінімізувати методом карт Карно ЛФ чотирьох логічних змінних, яку задано одиничним набором,

$$F_3 \{1,2,3,4,5,10,11,13\}.$$

Під одиничним набором розуміємо метод подання ЛФ у ДДНФ, при якому вказуються номери членів ПФ, які перетворюють її на одиницю. На основі одиничного набору складемо таблицю істинності ЛФ F_3 (табл. 1.3).

Таблиця 1.3

N п/п	X4	X3	X2	X1	F ₃
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	0

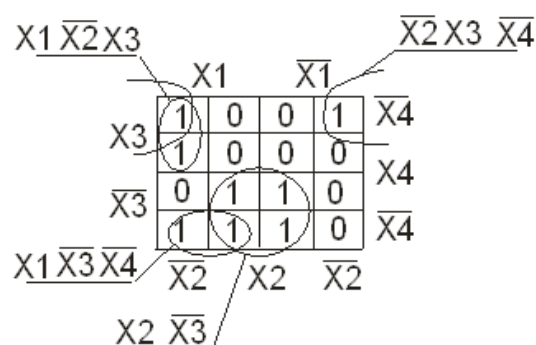


Рис. 1.3. Складання карти Карно функції $F_3(x_1, x_2, x_3, x_4)$

Мінімальна диз'юнктивна форма логічної функції:

$$F_3 = \overline{x_4}(\overline{x_2}x_3 + x_1\overline{x_3}) + x_2\overline{x_3} + x_1\overline{x_2}x_3.$$

У прикладі на рис. 1.3 кількість контурів – чотири, логічні імена контурів – $\overline{x_2x_3}, \overline{x_1x_3x_4}, \overline{x_2x_3x_4}, \overline{x_1x_2x_3}$.

Мінімізація частково визначених функцій. Частково визначеною функцією є така, значення якої задано не для всіх наборів аргументів. Це означає, що деякі набори аргументів є неможливими або недопустимими в реальних умовах. У таблиці істинності й карті Карно такі набори позначають знаками невизначеності X або ~. Якщо ці набори не існують, то ЛФ можна визначити з міркувань її найкращої мінімізації, тобто проставити у клітинах 0 або 1 замість знака невизначеності.

Мінімізація у монофункціональних базисах. Основою для отримання мінімальної форми в монофункціональному базисі (Шеффера або Пірса) є тупикова форма СДНФ. Метою переходу до монофункціонального базису є витяг із цієї форми однієї з операцій $x_1 + x_2$ та $x_1 \cdot x_2$. Робиться це так: для переходу в базис Шеффера (I–НЕ) необхідно взяти подвійне заперечення всього виразу, а для переходу в базис Пірса (АБО–НІ) – подвійне заперечення кожної імпліканти й загального виразу в цілому та застосувати закон Моргана – Шеннона.

Нехай $y = \overline{x_1x_2} + \overline{x_1x_2x_3} + \overline{x_3x_4}$, тоді

$$\overline{y_{ш}} = \overline{\overline{x_1x_2} + \overline{x_1x_2x_3} + \overline{x_3x_4}} = \overline{\overline{x_1x_2}} \cdot \overline{\overline{x_1x_2x_3}} \cdot \overline{\overline{x_3x_4}};$$

$$\overline{y_{п}} = \overline{\overline{\overline{x_1x_2}} + \overline{\overline{x_1x_2x_3}} + \overline{\overline{x_3x_4}}} = \overline{\overline{\overline{x_1}} + \overline{\overline{x_2}} + \overline{\overline{x_1}} + \overline{\overline{x_2}} + \overline{\overline{x_3}} + \overline{\overline{x_3}} + \overline{\overline{x_4}}}.$$

Приклад 2. Синтезувати методом Квайна однорозрядний повний суматор (рис. 1.4), заданий таблицею істинності однорозрядного повного суматора (табл. 1.4), де A, B – вхідні кількості A та B; P_i – вхід переповнення, P_0 – вихід переповнення, Σ – вихід суми A + B.

Таблица 1.4

A	B	P_i	Σ	P_0	$\overline{P_0}$
0	0	0	0	0	1
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	0

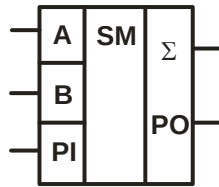


Рис. 1.4. Однорозрядний повний суматор (УГО)

Порядок синтезу

1. Запишемо логічні рівняння функцій P_0 та $\Sigma(A, B, P_i)$ у ДДНФ.

$$SM = \begin{cases} P_0 = \bar{A}BP_i + A\bar{B}P_i + AB\bar{P}_i + ABP_i; \\ \Sigma = \bar{A}\bar{B}P_i + \bar{A}B\bar{P}_i + A\bar{B}\bar{P}_i + ABP_i. \end{cases} \quad (1.5)$$

2. Проведемо спільну аналітичну мінімізацію отриманих виразів:

$$SM = \begin{cases} P_0 = \bar{A}BP_i + A\bar{B}P_i + AB\bar{P}_i + ABP_i; \\ \Sigma = \bar{A}\bar{B}P_i + \bar{A}B\bar{P}_i + A\bar{B}\bar{P}_i + ABP_i. \end{cases} \quad (1.5)$$

$$SM = \begin{cases} P_0 = (\bar{A}BP_i + ABP_i) + A\bar{B}P_i + AB\bar{P}_i + AB\bar{P}_i = BP_i + AP_i + AB; \\ \Sigma = \bar{A}(\bar{B}P_i + B\bar{P}_i) + A(\bar{B}\bar{P}_i + BP_i) = A \oplus B \oplus P_i. \end{cases} \quad (1.6)$$

Суматор можна реалізувати як два ЛБ, що вирішують незалежні завдання у вигляді схеми (рис. 1.5, а), а можна виконати, наприклад, у вигляді системи рівнянь (рис. 1.5, б):

$$SM \begin{cases} P_0 = BP_i + AP_i + AB; \\ \Sigma = (\bar{P}_0 + ABP_i) \cdot \overline{\bar{A}B\bar{P}_i} = (\bar{P}_0 + ABP_i) \cdot (A + B + P_i). \end{cases} \quad (1.7)$$

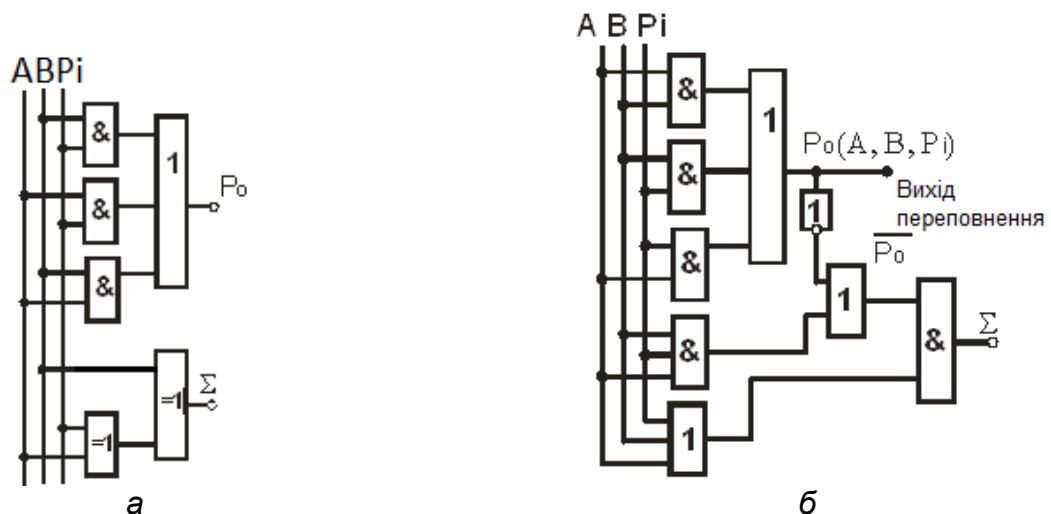


Рис. 1.5. Функціональна схема реалізації повного однорозрядного суматора: а – відповідно до системи рівнянь (1.6); б – відповідно до системи рівнянь (1.7)

Логічне рівняння для суми $\Sigma(A, B, P_i)$ отримано з інверсної функції $P_0(A, B, P_i)$ (див. табл. 1.4), у яких не збігаються верхній і нижній рядки: у верхньому рядку в сумі має бути 0, а $!P_0 = 1$, а в нижньому – навпаки. Тому нижній рядок виправляємо додаванням виразу ABP_i , а верхній – множенням на інверсію нульового рядка $(A + B + P_i)$. Реалізацію повного одноразрядного суматора за схемою, що відповідає системі (1.7), показано на рис. 1.5, б.

З порівняння обох схемних реалізацій випливає, що і за швидкістю і за кількістю елементів конструкції реалізація на рис. 1.5, б поступається реалізації, поданій на рис. 1.5, а. Однак, метод, який полягає в реалізації нової функції з фрагментів попередньої, може бути цілком виправданим під час реалізації інших, складніших ПФ.

У прикладі використовувалися такі методи аналітичної мінімізації:

1) склеювання

$$\overline{C}AB + ACB = AB(C + \overline{C}) = AB; \quad (1.8)$$

2) аксіома множини

$$\begin{aligned} X + X + \dots + X &= X, \\ X \cdot X \cdot \dots \cdot X &= X; \end{aligned} \quad (1.9)$$

3) теорема де Моргана

$$\begin{aligned} \overline{A + B} &= \overline{A} \cdot \overline{B}, \\ \overline{A \cdot B} &= \overline{A} + \overline{B}; \end{aligned} \quad (1.10)$$

4) закон поглинання

$$\begin{aligned} \overline{A} + AB &= \overline{A} + B = \\ &= \overline{\overline{\overline{A} + AB}} = \overline{\overline{A} \cdot \overline{AB}} = \\ &= \overline{A(\overline{A} + \overline{B})} = \overline{A \cdot \overline{A} + A \cdot \overline{B}} = \\ &= \overline{0 + A \cdot \overline{B}} = \overline{A \cdot \overline{B}} = \overline{A} + B. \end{aligned} \quad (1.11)$$

2. ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ C

Пріоритетність виконання операторів у виразах мови C наведено в табл. 2.1.

Таблица 2.1

Рівень	Оператори	Категорія	Опис
1	()		Круглі дужки
	[]		Елемент масиву
		Доступ до даних	Звернення до елемента структури, наприклад: <code>PORTB.1</code> — розряд 1 порта B

Закінчення табл. 2.1

	->		Звернення до елемента структури, визначеної покажчиком, наприклад: pStruct->x — елемент x структури, на яку вказує pStruct
Рівень	Оператори	Категорія	Опис
	++, -- (постфікси)	Арифметичні	Оператори автоінкремента й автодекремента після того, як вираз, у якому задіяно відповідні операнди, обчислено. Приклади: a = b++; рівнозначно a = b; b = b+1; a = b--; рівнозначно a = b; b = b -1;
2	++, -- (префікси)	Арифметичні	Оператори автоінкременту й автодекременту перед тим, як вираз, у якому задіяно відповідні операнди, буде обчислено. Приклади: a = ++b; рівнозначно b = b+1; a = b; a = --b; рівнозначно b = b-1; a = b;
	!	Логічні	Логічне (унарне) заперечення
	~	Порозрядні	Порозрядне заперечення
	&	Доступ до даних	Адреса
3	+, - (унарні)	Арифметичні	Змінення знака операнда
	* (унарні)	Доступ до даних	Розіменування покажчика
4	* (бінарні)	Арифметичні	Множення
	/	Арифметичні	Ділення
	%	Арифметичні	Остача від ділення
5	+, - (бінарні)	Арифметичні	Додавання й віднімання
6	«	Порозрядні	Порозрядний зсув уліво
	>>		Порозрядний зсув управо
7	<, >, <=, >=	Порівняння	Менше, більше тощо
8	==, !=	Порівняння	Дорівнює
9	&	Порозрядні	Порозрядне «І»
10	^	Порозрядні	Порозрядне «Виключне АБО»
11	I	Порозрядні	Порозрядне «АБО»
12	&&	Логічні	Логічне «І»
13		Логічні	Логічне «АБО»
14	=	Присвоєння	Можливі також поєднання оператора присвоєння з арифметичними й порозрядними операторами, наприклад: a += b; рівнозначно a = a + b; a *= b+c; рівнозначно a = a * (b + c) ;

Об'єднані за деякою ознакою послідовності виразів беруть у фігурні дужки {}. Так, наприклад, позначаються межі функцій, а також блоки виразів у циклічних та умовних конструкціях.

2.1. Структура програми мовою C

Структура програми, написана мовою C*, має такий вигляд:

```
// Директиви препроцесора
#include "time_delay.h"
// Оголошення глобальних типів, змінних і констант
...
//Функції
Function1
{
    // Оголошення локальних типів, змінних і констант
    ...
    //Оператори
    ...
}
...
FunctionN
{
    // Оголошення локальних типів, змінних і констант
    ...
    //Оператори
    ...
}
// Головна функція програми
int main (void)
{
    // Оголошення локальних типів, змінних і констант
    ...
    //Оператори
    ...
}
```

Програми мовою C зазвичай починаються з **директив препроцесора** (починаються із символу «#»), який по суті не є конструкцією мови C та обробляється до фактичної компіляції програми. Їх призначення — підставлення деякого коду в програму. Так, наприклад, дуже часто використовується директива *#include*, яка включає у файл з вихідним кодом

програми текст зовнішнього заголовного файлу (з розширенням *.h*). **Заголовні файли** містять означення глобальних типів, констант, змінних і функцій.

2.2. Типи даних, змінні, константи

Тип даних визначає діапазон допустимих значень і простір, який відводиться в пам'яті даних, для змінних, констант і результатів, що повертаються функціями. У різних компіляторах з мови можуть використовуватися власні типи даних, проте всі вони базуються на стандартних типах, перелічених у табл. 2.2.

Таблиця 2.2

Тип	Назва типу даних	Розмір, біти	Діапазон значень
bit	Біт	1	0, 1
char	Символ	8	-128...127
int	Ціле кількість	16	-32768...32767
float	Дійсне кількість	32	$\pm 1,175 \cdot 10^{-38} \dots \pm 3,402 \cdot 10^{38}$
long int	Довге ціле	32	-2147483648..2147483647
unsigned char	Беззнаковий символ, або логічне значення	8	0...255 TRUE, FALSE
unsigned int	Беззнакове ціле	16	0...65535
unsigned long int	Беззнакове довге ціле	32	0...4294967295

Розглянемо деякі означення докладніше, щоб зрозуміти, як утворюються перелічені в табл. 2.2 діапазони допустимих значень.

Біт — це базова одиниця інформації в обчислювальній техніці, що означає один із двох станів: 0 або 1, високий або низький рівень сигналу. Наприклад, під час під'єднання світлодіодів до виводів мікроконтролера для кожного такого виводу програміст оперує поняттям біта, оскільки в цьому випадку йдеться про два рівні напруги, один з яких вмикає, а інший вимикає світлодіод. Загалом кажуть, що біт містить логічний нуль або логічну одиницю.

Наступна фундаментальна одиниця інформації – байт, що складається з вісьми бітів. З його допомогою можна подати 256 різних станів (0...255), що наведено в табл. 2.3.

Таблиця 2.3

Значення	Двійкове подання	Вісімкове подання	Шістнадцяткове подання
0	0b00000000	0000	0x00
1	0b00000001	0001	0x01
2	0b00000010	0002	0x02
Значення	Двійкове подання	Вісімкове подання	Шістнадцяткове подання
3	0b00000011	0003	0x03
4	0b00000100	0004	0x04
5	0b00000101	0005	0x05
6	0b00000110	0006	0x06
7	0b00000111	0007	0x07
8	0b00001000	0010	0x08
9	0b00001001	0011	0x09
10	0b00001010	0012	0x0A
11	0b00001011	0013	0x0B
12	0b00001100	0014	0x0C
13	0b00001101	0015	0x0D
14	0b00001110	0016	0x0E
15	0b00001111	0017	0x0F
16	0b00010000	0020	0x10
...	...	...	...
254	0b11111110	0376	0xFE
255	0b11111111	0377	0xFF

2.3. Правила перетворення однієї системи числення на іншу

Схема перетворення деякої системи числення на десяткову є дуже простою: кожен цифру числа множимо на основу системи, піднесену до відповідного розряду степеня (починаючи з 0), а потім складаємо отримані результати для всіх розрядів. Приклад для десяткового числа 100:

$$\begin{aligned}
 0b01100100 &= 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = \\
 &= 0 + 64 + 32 + 0 + 0 + 4 + 0 + 0 = 100; \\
 0144(8) &= 1 \cdot 8^2 + 4 \cdot 8^1 + 4 \cdot 8^0 = 64 + 32 + 4 = 100; \\
 0x64 &= 6 \cdot 16^1 + 4 \cdot 16^0 = 96 + 4 = 100.
 \end{aligned}$$

Для перетворення десяткової системи числення на іншу число слід розділити на основу системи, запам'ятати остачу, потім результат ще раз розділити на основу системи, знову запам'ятати остачу і так далі доти, доки не буде отримано неподільну частку, яка є старшим розрядом отриманого подання, інші розряди формуються із остач, починаючи від останньої до першої.

Приклад перетворення числа 100 на двійкову систему числення:

100/2	=	50,	остача 0
50/2	=	25,	остача 0
25/2	=	12,	остача 1
12/2	=	6,	остача 0
6/2	=	3,	остача 0
3/2	=	1,	остача 1

1 на 2 не ділиться. Результат: 100 = 0b1100100.

Приклад перетворення числа 100 на вісімкову систему числення:

100/8	=	12,	остача 4
12/8	=	1,	остача 4

1 на 8 не ділиться. Результат: 100 = 0144.

Приклад перетворення числа 100 на шістнадцяткову систему числення:

100/16	=	6,	остача 4
--------	---	----	----------

6 на 16 не ділиться. Результат: 100 = 0x64.

Тип char

Байту даних у мові C відповідає тип **char**, який дістав свою назву від англійського слова *character*, оскільки найчастіше використовується в програмах для зберігання символів (тобто їх ASCII-кодів). Однак з його допомогою можна звертатись і до невеликих цілих чисел у діапазоні від -128 до +127, а для типу *unsigned char* – від 0 до 255, оскільки в цьому випадку знаковий розряд не використовується (використання слова *unsigned* (беззнакове) має такий же зміст і у випадку цілого типу *int*).

Примітка. У багатьох компіляторах з мови C замість типу *unsigned char* використовується його еквівалент виду *byte* або *Byte*.

Типи користувача

Мова C дає змогу крім стандартних оголошувати власні типи. Для цього використовується ключове слово *typedef*, наприклад:

```
typedef unsigned char byte; // оголошуємо тип byte
typedef unsigned int word; // оголошуємо тип word
```

Іншими словами, для оголошення типу користувача використовується конструкція вигляду

```
Typedef стандартний_тип_ідентифікатор_користувацького_типу;
```

Так, у компіляторі WinAVR визначено такі типи користувачів:

```
typedef signed char int8_t — у заголовному файлі int types.h;
typedef unsigned char uint8_t — у заголовному файлі int -
types.h;
typedef int int16_t — у заголовному файлі inttypes.h;
typedef unsigned int uint16_t — у заголовному файлі
inttypes.h;
typedef long int32_t — у заголовному файлі inttypes.h;
typedef unsigned long uint32_t — у заголовному файлі
inttypes.h;
typedef long long int64_t — у заголовному файлі inttypes.h;
typedef unsigned long long uint64_t — у файлі inttypes.h;
typedef struct {int quot; int rem; } div_t; — у заголовному
файлі stdlib.h (цей тип використовується стандартною
функцією div ());
typedef struct {long quot; long rem;} ldiv_t; —В у заголовному
файлі stdlib.h (цей тип використовується стандартною
функцією ldiv ( ) ).
```

Змінні

Змінна — це іменована величина певного типу, яка може змінюватися під час виконання програми. Для оголошення змінних (тобто виділення їм пам'яті) у програмі мовою C використовується така конструкція:

```
тип_змінної ідентифікатор1, ідентифікатор 2, ...;
```

Наприклад:

```
int i; // Оголошення цілої змінної i
char c1, c2; // Оголошення символьних змінних c1 і c2
```

Примітка. Тип користувача, що використовується під час оголошення змінної, має бути оголошено вище в тексті програми.

Для доступу до змінної програми використовується відповідний ідентифікатор (обов'язково після оголошення змінної). Значення, що надаються змінною, мають відповідати їй за типом (або правилами наведення типів, що розглядаються нижче). Приклади:

```
i = 2;    // Помилка! Змінну i ще не оголошено
int i;    // Оголошення цілої змінної i
float f;  // Оголошення дійсної змінної f
i = 2;    // Змінній i надано значення 2
```

```
f = 3.3 // Змінній f надано значення 3.3
f = i;  // Змінній f надано значення змінної i
        //( у цьому випадку буде виконано автоматичне
        // наведення типів, тобто f = 2.0)
```

За доступністю змінні поділяються на глобальні й локальні. До **глобальних змінних** мають доступ усі функції програми. Такі змінні оголошуються в програмі перед оголошенням усіх функцій. До **локальних змінних** має доступ тільки функція, у якій їх оголошено.

Межі видимості змінних

Імена змінних мають певну *межу видимості*, під якою розуміють, що компілятор використовує змінні відповідно до того, де вони знаходяться. Імена змінних, оголошені всередині функції, мають межу видимості, обмежену конкретною функцією. Наприклад, у кількох функціях можна оголосити змінну `int i`, яка в кожній функції не матиме жодного зв'язку з аналогічними змінними в інших функціях. Так само і змінна, оголошена всередині блока (який обмежено фігурними дужками `{...}`), залишається локальною відносно цього блока.

Глобальні змінні мають межу видимості, які починаються від місця їх оголошення і тривають до завершення програмного файлу. Щоб глобальну змінну можна було використовувати в інших файлах, її потрібно оголосити з допомогою ключового слова `extern`:

```
extern int n;
```

Оголошену таким чином змінну перед її використанням, слід обов'язково ініціалізувати в зовнішньому файлі деяким значенням.

Константи

Константа — це іменована величина певного типу, яка на відміну від змінної не може змінюватися під час виконання програми, а має конкретне значення, обчислене на момент оголошення. Для оголошення констант у програмі C використовується така конструкція:

```
const тип_константи ідентифікатор = значення;
```

Наприклад:

```
const int i = 10; // Оголошення цілої константи i
```

Примітка. Тип користувача, який застосовується при оголошенні константи, має бути оголошено вище в тексті програми.

Величину, оголошену як константу, буде розміщено компілятором у пам'яті програм, а не в обмеженій області змінних RAM.

Для доступу до константи використовується її ідентифікатор.

Приклади:

```
c = 'A';           // Помилка! Константа ще не оголошена
int i = 2;         // Оголошення цілої змінної i; Змінній i
                  // присвоєно початкове значення 2
const c = 'A';     // Оголошення константи c
c = i;             // Помилка! Спроба присвоїти значення
                  // константі
i = c;             // Змінній i присвоєно значення константи c.
                  // У цьому випадку буде виконано автоматичне
                  // наведення типів, тобто i = 65 (код ASCII відповідає
                  // символу 'A')
```

Примітка. У мікроконтролерах AVR використовується роздільна пам'ять для даних (SRAM) і програм (Flash), а також пам'ять типу EEPROM. Змінні, якщо не зазначено іншого, за замовчуванням розміщуються в пам'яті SRAM, але якщо трапляється слово *const*, компілятор відносить відповідні значення до постійної флеш-пам'яті. Крім того, компілятор WinAVR дає змогу записувати константи в пам'ять EEPROM і зчитувати їх з допомогою відповідних функцій.

Перелічені типи

Перелічений тип — це оголошення списку цілих констант, які можна явно не ініціалізувати (у цьому випадку компілятор вважає, що перша константа в списку набуває значення 0, друга – 1 і т. д.). Для такого оголошення використовується ключове слово *enum*:

```
int n;
enum (zero, one, two);    //zero = 0; one = 1; two = 2
n = one;                  //n = 1
```

Якщо необхідно змінити початкове значення для списку констант, то можна вказати його явно під час оголошення, наприклад:

```
enum (three = 3, four, five); //three = 3; four = 4; five = 5
```

Зведення типів

Зведення типів — це примусове перетворення значення одного типу

на інше, сумісне з вихідним. Це важливо під час виконання арифметичних операцій, коли отримані значення можуть виходити за допустимі межі.

Зведення типів може бути явним і неявним. Неявне зведення типів використовується в операторах, коли компілятор сам виконує необхідні перетворення без участі програміста. Приклади такого зведення вже розглядалися вище в підрозділах, які стосуються змінних та констант.

Для явного зведення типу деякої змінної перед нею слід зазначити у круглих дужках ім'я нового типу, наприклад:

```
int X;  
int Y = 200;  
char C = 30;  
X=(int)C*10+Y; // Змінна C зведена до типу int
```

Якби в цьому прикладі не було виконано явне зведення типів, то компілятор припустив би, що вираз $C * 10$ – це вісьмизрядне множення (розрядності типу `char`) і замість коректного значення 300 (0x12C) у стек було би поміщено зрізане значення 44 (0x2C). Таким чином, унаслідок обчислення виразу $3*10+Y$ змінній `X` було б присвоєно значення 640, а не коректне 3200. Унаслідок зведення типу змінна розпізнається компілятором як 16-и розрядна, і описаної вище помилки не виникає.

Оператор `sizeof`

Оператор `sizeof` застосовується для обчислення розміру області пам'яті (у байтах), яка відводиться під деяку змінну, результат виразу або тип.

Наприклад:

```
int a;  
float f;  
a = sizeof(int); //a = 2  
a = sizeof(f);   //a = 4  
f = 3.3;  
a = sizeof(f + a); //a = 4, оскільки тип результату - float
```

Функції

Функція є «контейнером», у якому виконується деякий фрагмент програмного коду. Використання функцій спрощує написання й налаштування програм, оскільки в них зручно розміщувати повторювані групи операторів. Будь-яка програма мовою C містить головну функцію під назвою `main()`.

Ця функція під час запуску програми виконується першою.

Функції користувача визначаються після директив препроцесора і глобальних оголошень типів, змінних і констант у файлі з вихідним кодом або в заголовному файлі. При цьому використовується такий синтаксис:

```
Тип_значення, що повертається, _Ім'я_функції (Список параметрів)
{
    //Тіло функції
}
```

Примітка. Попереднього оголошення функції може не бути. У цьому випадку вона є доступною тільки всередині файлу, у якому її визначено.

Як тип значення, що повертається, може використовуватися ключове слово `void`. Це означає, що функція не повертає ніякого значення (у деяких мовах програмування такі функції називають процедурами).

Функцію викликають за її іменем із зазначенням у круглих дужках переліку параметрів, що передаються (якщо їх немає, то в дужках нічого не вказується), наприклад:

```
void Function1(int n, char c)
{
...
{
int Function2()
}
...
{
int main()
{
int x; char y;
Function1(x, y);
x = Function2();
}
```

Параметри — це ідентифікатори, які можна використовувати всередині функції. Замість них підставляють відповідні значення, указані під час виклику функції (якщо у функцію передається більше одного параметра, то їх відокремлюють один від одного комами під час як оголошення, так і виклику).

Значення, передані у функцію, фактично не змінюються, а просто копіюються в параметри, які мають значення локальних змінних. При цьому слід стежити за тим, щоб тип значень, що передаються, відповідав типу параметрів, оголошених у заголовку функції.

Повертані значення

Якщо функцію призначено для повернення даних певного типу, то для цього в її тілі використовують ключове слово `return`, після якого (через пропуск) указують повертане значення. При цьому всі оператори після слова `return` ігноруються і відбувається повернення даних у викликальну функцію, наприклад:

```
int power3(int n)
{
    return n*n*n;
}
void main()
{
    int x;
    x = power3(2); //x = 8
}
```

Слово `return` можна також використовувати без зазначення повертаного виразу, яке в цьому випадку просто означає вихід із функції.

Прототипи функцій. У звичайному варіанті функції використовують лише після їх визначення, проте бувають випадки, коли функції викликають одна одну, і організувати їх «правильне» визначення неможливо. Обійти цю проблему дають змогу прототипи функцій, які є оголошенням до визначення. Таке оголошення є лише заголовком функції, причому в списку параметрів указують тільки типи (без ідентифікаторів), наприклад:

```
int f1(int); void f2(int, int);
int f1(int x)
{
}
void f2(int a, int b)
```

Прототипи функцій часто використовують в заголовних файлах, що включаються до тексту програми з допомогою директиви препроцесора `#include`.

Класи пам'яті при оголошенні локальних змінних

Локальні змінні може бути оголошено всередині функцій як такі, що належать до одного з трьох класів пам'яті:

- **auto** (значення за замовчуванням, можна явно не вказувати) – при оголошенні змінна не ініціалізується жодним значенням (значення –

поточний вміст області пам'яті, відведеної під змінну); під час виходу з функції змінна видаляється з пам'яті;

- **static** — статична змінна є доступною лише в межах функції, хоча пам'ять виділяється для неї в просторі глобальних змінних; під час першого звернення до функції ініціалізується нульовим значенням і після виходу з функції не видаляється (отже, під час наступних звернень до функції в ній міститься попереднє значення);

- **register** — аналог автоматичної локальної змінної, за винятком того, що компілятор може виділити для неї не область пам'яті даних, а робочий регістр мікроконтролера, що дуже прискорює звернення до значення змінної.

Приклад використання статичної змінної:

```
...
int plus5()
{
    static int x;
    return x + 5;
}
void main()
{
    int y;
    y = plus5(); //y = 5
    y = plus5(); //y = 10
    y = plus5(); //y = 15
}
```

Рекурсія

Рекурсія — це виклик функцією самої себе. Ця можливість буває важкою в розумінні, і тому не дивно, що експерти з мови так люблять рекурсивні функції. Приклад рекурсивного виклику:

```
void fl(int n)
{
    int x;
    fl(x);
}
```

Незважаючи на складність сприйняття, рекурсія часто використовується в стандартних бібліотечних функціях, а також у багатьох алгоритмах сортування. Однак під час програмування мікроконтролерів використання рекурсії зазвичай загрожує проблемами через обмежений обсяг оперативної пам'яті. Йдеться про те, що з кожним викликом рекурсивної функції частина пам'яті витрачається для збереження даних у

стек. Ці дані зберігаються там, поки не буде виконано повернення з функції. Отже, коли рекурсивна функція знову й знову викликає саму себе, у стеку залишається дедалі менше вільного міста.

Можна сказати, що в переважній більшості випадків використання рекурсії під час програмування мікроконтролерів – це надійний спосіб швидкого й непередбачуваного заповнення стеку. Це часто призводить до виникнення помилкового стану, який називають переповненням стеку. Область стеку зазвичай розміщується у верхній частині пам'яті й збільшується «вниз», тоді як область змінних розміщується в нижній частині пам'яті й збільшується «вверх». Тому якщо обсяг даних, що містяться в стеку, перевищить розмір області стеку, то ці дані можуть досягти області змінних і затерти їх значення. При цьому помилку переповнення стеку не завжди легко виявити, оскільки вона може виникати лише періодично.

Примітка. Така проблема може виникнути також при багаторазовому вкладенні функцій, тобто коли функція `f1` викликає функцію `f2`, яка викликає функцію `f3`, яка викликає функцію `f4` і т. д.

Структури

Структура — це спеціальний тип даних, що складається з кількох різнотипних змінних (полів). У випадку оголошення структура має такий вигляд:

```
struct ім'я_структури {  
    тип поле_1;  
    ...  
    тип поле_N;  
};
```

Як і інші типи, структуру можна і в подальшому використовувати для оголошення змінних, наприклад:

```
struct MyStructure { // Оголошення структури MyStructure  
    int Field1;  
    char Field2;  
    float Field3;  
};  
// Оголошення змінних YourStruct и OurStruct типу  
MyStructure  
struct MyStructure YourStruct, OurStruct;
```

Допускається ініціалізація полів безпосередньо під час оголошення змінних структур з допомогою переліку значень у фігурних дужках,

наприклад:

```
struct DATE {
    int Day;
    int Month;
    int Year;
}
struct DATE MyBirthday = (22, 8, 1994);
```

Для доступу до полів структури у програмі використовують запис типу «Структури. поле», тобто у наведеному вище прикладі структури DATE для ініціалізації полів можна було скористатися такими операторами:

```
MyBirthday.Day = 22;
MyBirthday.Month = 8;
MyBirthday.Year = 1994;
```

Структури, зі свого боку, можуть бути полями інших структур, наприклад:

```
struct ME {
    char MyName[30]; // Рядок завдовжки 30 символів – ім'я
    struct DATE MyBirthday; // Структура, що зберігає день
    народження}
...
struct ME MyData;
MyData.MyName = "Jason Bern";
MyData.MyBirthday.Day = 22;
MyData.MyBirthday.Month = 8;
MyData.MyBirthday.Year = 1994;
```

Структури можуть бути параметрами функцій, а також результатом, що повертається. Приклад функції, що повертає структуру типу DATE :

```
struct DATE January1(int CurYear)
{
    struct DATE Jan01;
    Jan01.Day = 1;
    Jan01.Month = 1;
    Jan01.Year = CurYear;
    return Jan01;
}
...
struct DATE BeginOfTheYear;
BeginOfTheYear = January1(2019) ;
```

Показчики на адреси змінних

Показчик — це змінна, що містить адресу деякого елемента даних (змінної, константи, функції, структури). У мові C показчики тісно пов'язані з обробленням масивів і рядків, які буде розглянуто далі. Для оголошення змінної як показчика використовується оператор *:

```
int *p; //p - показчик на ціле число
```

Для присвоєння адреси деякій змінній показчику використовується оператор &, наприклад:

```
char *p; // показчик на символ  
char c; // символна змінна  
c = ' A' ;  
p = &c; // p містить адресу змінної c (указує на 'A')
```

Для отримання значення змінної, на яку вказує показчик, використовується оператор розіменування *.

```
char *p;  
char c, b;  
c = 'A';  
p = &c ;  
b = *p; //Тепер b = 'A'
```

Цей оператор можна також використовувати й для присвоєння деякого значення змінній, на яку вказує показчик:

```
char *p;  
char c, b;  
b = ' A' ;  
p = &c;  
*p = b; //Тепер c = 'A'
```

Стосовно програмування мікроконтролерів показчики можна використовувати, наприклад, для запису даних у порт введення/виведення. Припустимо, регістр даних порту розташований у пам'яті за адресою 0x16. У цьому випадку для запису в нього значення 0xFF можна скористатися таким фрагментом програмного коду:

```
unsigned char *thePort;  
thePort = 0x16;  
*thePort = 0xFF;
```

Передача до функцій параметрів за посиланням

З допомогою покажчиків, які використовуються як параметри функцій, можна організувати повернення більше одного значення. Розглянемо такий приклад:

```
int SumAndDiv(int *a, int *b)
{
    int bufA, bufB;
    bufA = *a;
    bufB = *b;
    *a = bufA / bufB; //Покажчик посилається на результат
                     //цілісного ділення без остачі
    b = bufA % bufB; //Покажчик посилається на остачу від
                     //цілісного ділення
    return bufA + bufB; //Функція повертає суму
}

...
int v1, v2, sum;
v1 = 10;
v2 = 3;
sum = SumAndDiv(&v1, &v2); //sum = 13; v1 = 3; v2 = 1
```

У функції `SumAndDiv` спочатку зберігаються в буферних змінних значення, на які вказують покажчики `a` і `b`. Потім у змінну, на яку вказує покажчик `a`, записується результат ділення переданих у функцію значень без остачі, а в змінну, на яку вказує покажчик `b`, – остача від ділення. Оскільки з допомогою покажчиків виконувалося безпосереднє звертання до комірок пам'яті, а не змінних, то після виходу з функції вміст цих комірок залишається незмінним. Під час виклику функції `SumAndDiv` до неї передаються адреси змінних `v1` і `v2`, яким у тілі функції відповідають покажчики `a` і `b`.

Покажчики на структури

Для структур можна створювати покажчики точно так, як і будь-якого іншого типу даних. Приклад застосування такого покажчика:

```
struct DATE {
    int Day;
    int Month;
    int Year;
}
```

```
...
struct DATE MyBirthday, *dateP;
dateP = &MyBirthday; //dateP указує на структуру
MyBirthday
```

У такому разі для доступу до полів структури через покажчик використовується оператор ->:

```
dateP->Day = 22;
dateP->Month = 8;
dateP->Year = 1994;
```

Можна також використовувати і розіменування покажчика:

```
(*dateP) .Day = 22 ;
(*dateP) .Month = 8;
(*dateP) .Year = 1994;
```

Примітка. В останньому прикладі операцію *dateP узято в дужки, оскільки оператор розіменування * має менший пріоритет порівняно з оператором доступу до поля структури.

Покажчики можуть бути полями структури. Це часто використовується для створення в пам'яті ланцюжків однотипних структур, коли кожна попередня вказує на наступну:

```
struct DATE {
    int Day;
    int Month;
    int Year;
    struct DATE *next date; // покажчик на іншу структуру
                           //того ж типу
}
...
struct DATE date1, date2;
date1.next_date = &date2;
```

Тепер, наприклад, оператору date1. next_date->Year відповідає доступ до поля Year структури date2 (тобто це є еквівалентним запису date2. Year).

Масиви й рядки

Масив — це тип даних, що використовується для подання

послідовності однотипних значень. Масиви оголошуються подібно до звичайних змінних із зазначенням у квадратних дужках розмірності (кількості елементів у масиві):

```
int digits[10]; // Масив із десяти елементів типу int
char str[10]; // Масив із десяти символів (рядок)
```

Доступ до елементів масиву реалізується з допомогою індексу (порядкового номера елемента, починаючи з 0):

```
digits[0] = 0;
digits[1] = 1;
str[0] = 'A';
str[1] = 'B';
```

Примітка. Власне, ім'я масиву — це покажчик на його перший елемент. Так, в останньому прикладі оператор `str[0] = 'A';` можна було б замінити оператором `*str = 'A';`, а оператор `str[1] = 'B';` — оператором

```
*(str + 1) = 'B';
```

Найчастіше набагато зручніше ініціалізувати масив безпосередньо під час його оголошення, наприклад:

```
int digits [10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char str [10] = {'T', 'h', 'e', ' ', 'l', 'i', 'n', 'e'};
int n;
char c;
n = digits[2]; //n = 2
c = str[1]; //c = 'h'
```

Під час розроблення програм слід урахувувати, що компілятор не завжди може відстежити вихід індексу за межі масиву. Іншими словами, якби в наведеному вище прикладі використовувався оператор `c = str[n+10];`, то на етапі компіляції помилки б не виникало, але під час виконання програми змінній `c` буде присвоєно значення 13-го елемента масиву, якого не існує, тобто зчитано вміст області пам'яті, адресу якої на 12 елементів зміщено відносно початкової адреси масиву.

Рядки

Рядок у мові C — це масив типу `char`. Вище було розглянуто приклад оголошення такого масиву, що відповідає рядку `The Line`. Це оголошення

можна було б виконати і з допомогою рядкового літералу:

```
char str[10] = "The line";
```

Однак у такому випадку слід пам'ятати, що компілятор неявно завершує рядкові літерали символом '0', отже, реальна довжина рядка буде більшою на 1. Це слід урахувати, щоб під час ініціалізації масиву не вийти за його межі.

Під час ініціалізації рядків розмірність масиву можна не вказувати:

```
char str[] = "The Line";
```

Тут компілятор визначить розмірність самостійно (це правило стосується й ініціалізації будь-яких інших масивів).

Щоб рядкові константи розміщувалися не в пам'яті SRAM, а лише у флеш-пам'яті мікроконтролера, їх слід оголошувати з допомогою слова

```
const: const char str[].
```

Багатовимірні масиви

У мові C допускається використання багатовимірних масивів, тобто масивів, елементами яких є масиви. Приклади оголошення таких масивів:

```
int a2[10][2];    // Двовимірний масив 10x2
int a3[3][2][5];  // Тривимірний масив 3x2x5
```

Фактично всі елементи багатовимірного масиву зберігаються в пам'яті послідовно, тому наведені приклади ініціалізації є аналогічними:

```
int a[2][3] = { {1, 2, 3},
                {4, 5, 6} };
i
int a [2] [3] = { 1, 2, 3, 4, 5, 6 };
```

Для доступу до елементів багатовимірного масиву використовуються індекси кожної з розмірностей або операції розіменування покажчика. Наприклад, щоб витягти в деяку змінну `n` цифру 4 з наведеного вище масиву `a`, можна скористатися одним із двох варіантів:

```
n = a[1][0]; // Виймаємо елемент із рядка 1 (другий),
             // стовпця 0 (перший), тобто цифру 4
n = *(a + 3); // a - покажчик на перший елемент масиву,
             // отже, a + 3 - покажчик на 4-й елемент
```

Умовні оператори

Умовні оператори використовуються для виконання того чи іншого блока, коли йдеться про залежність від деякої умови. До таких операторів у мові C належать `if-else` і `switch-case`.

Оператор *if-else*

У найпростішому випадку оператор `if-else` має таку структуру:

```
if (умовний_вираз) блок_кода_1; else блок_кода__2;
    case константа-вираз1 : блок_кода
    case константа-вираз2 : блок_кода
    case константа-вираз3 : блок_кода
    default: блок_кода
}
```

Продемонструємо використання оператора `switch-case` на прикладі розглянутого вище фрагмента, реалізованого з допомогою оператора `if-else`:

```
switch (a) {
    case 1 : b = a + 3; break;
    case 2 : b = a + 10; break;
    case 3 : b = 0; break;
    default:
}
```

Оператор `break` приводить до негайного виходу з блока `switch`. Якщо з якихось причин необхідно продовжити перевірку відповідності виразу вибору константам, указаними у гілках `case`, то слід видалити оператори `break`.

До того ж гілки `case` можна каскадувати. Такий прийом є особливо зручним у тих випадках, коли необхідно, наприклад, перевірити введений символ, не перевіряючи, з якої літери його написано, чи є він великою літерою чи малою, або коли необхідно отримати одну й ту ж реакцію на кілька різних чисел. Розглянемо це з прикладу фрагмента деякої абстрактної програми:

```
switch (input) {
    case 'a' : case 'A' : DoA(); break;
    case 'b' : case 'B' : DoB(); break;
    case '0' : case '1' : case '2' : case '3' : Do0123();
break;
    case '4' : case '5' : case '6' : case '7' : Do4567();
```

```
break;
    default: DoDefault(); break;
}
```

Циклічні конструкції

Циклічні конструкції застосовують для повторення деякого блока коду, на підставі з умови циклу. У мові C використовуються циклічні конструкції `while`, `for` і `do-while`.

Конструкція *while*

Цикл `while` має такий синтаксис:

```
while (умовний_вираз)
{
    // Виконання тіла циклу, якщо вираз є істинним
}
```

Іншими словами, цикл `while` має сенс використовувати у випадках, коли відповідний оператор або блок операторів виконується доти, доки умовний вираз є істинним. Приклад формування рядка, що складається з непарних цифр:

```
int c, i;
const char str[] = "0123456789";
char OddNums[5]; // Рядок для зберігання непарних цифр
c = 0;           // Лічильник циклів
i = 0;           // Індекс масиву OddNums
while (c < 10)   // Робити доти, доки c є меншим за 10, ...
{
    //Якщо остача від ділення на 2 становить 1, то записуємо
    //в i-ту позицію масиву OddNums c-й елемент рядка str, після
    //чого значення i автоматично інкрементується
    if ((c % 2) == 1) OddNums[i++] = str[c];
    c++;           //c = c + 1
}
```

Конструкція *for*

Цикл `for` має такий синтаксис:

```
for (вираз1; вираз2; вираз3)
{
    // Виконання тіла циклу
}
```

Вираз1 виконується тільки один раз під час входу в цикл і зазвичай є оператором присвоєння деякого початкового значення лічильнику циклу. Вираз2 — це умовний вираз, що визначає момент виходу з циклу (цикл виконується доти, доки результат виразу2 є TRUE або вираз 1). Вираз3 — ще один оператор присвоєння, у якому зазвичай змінюється лічильник циклу або деяка змінна, що впливає на виконання умови у виразі 2. Вирази можна подати будь-якими операторами, зокрема порожніми (тобто замість виразу можна поставити лише символ крапки з комою).

Цикли `while` і `for` в більшості випадків є взаємозамінними. Так, наведений вище приклад для циклу `while` можна переписати в такому вигляді:

```
int c, i;
const char str[] = "0123456789";
char OddNums[5]; // Рядок для зберігання непарних цифр
i = 0; //Індекс масиву OddNums
for (c=0; c < 10; C++) //Доти, доки c є меншим за 10, ...
    if ((c % 2) == 1) OddNums[i++] = str[c];
```

В одних ситуаціях зручніше застосовувати цикл `for`, в інших — `while`. Перевагою циклу `for` є наочна ініціалізація та організація змінення лічильника циклу. З іншого боку, цикл `while` є більш гнучким і забезпечує більше можливостей для реалізації нестандартних програмних рішень під час організації повторюваних обчислень.

Конструкція *do-while*

Крім циклів `while` і `for`, у яких спочатку виконується перевірка істинності умови циклу і потім керування передається блоку операторів циклу, у мові C є також конструкція `do-while`. Вона відрізняється від перших двох тим, що в ній спочатку виконується блок операторів і лише потім перевіряється виконання умови. Іншими словами, цикл `do-while` завжди виконується щонайменше один раз незалежно від умови циклу.

Цикл `do-while` має такий синтаксис:

```
do
{
    // Виконання блока операторів циклу
}
while (Умовний_вираз);
```

Організація нескінченних циклів

Для організації нескінченного циклу як умовний вираз у конструкції `while` або `do-while` можна просто вказати значення `TRUE` або `1`:

```
while(1) блок_операторів;
```

У випадку циклів `for` це буде мати такий вигляд:

```
for (;;) блок__операторів;
```

Оператори *break* і *continue*

Якщо в тілі будь-якого циклу трапляється оператор `break`, то керування відразу передається на оператор, що йде за оператором циклу, незалежно від істинності чи неістинності умовного виразу. При цьому в укладених циклах вихід здійснюється не на верхній рівень укладеності, а лише на один рівень уверх.

Під час виконання оператора `continue` всі оператори блока, що знаходяться після нього, пропускаються, а керування передається на початок циклу для наступної ітерації. На практиці оператори `continue` використовуються набагато менше, ніж оператори `break`, однак у складних циклах, що потребують прийняття рішень на основі багатьох факторів, використання операторів `continue` може бути дуже зручним.

Стандартні функції введення/виведення

Стандартні функції введення/виведення дають змогу обмінюватись інформацією з виконуваною програмою, що, наприклад, дуже зручно під час настроєння. Усіх їх побудовано на основі функцій посимвольного введення/виведення, які легко пристосувати до апаратних вимог системи. Якщо під час звичайного застосування мови C ці функції використовуються для введення даних з клавіатури й виведення на екран або принтер, то стосовно мікроконтролерів AVR йдеться про обмін даними через приймач `UART/USART` (за замовчуванням) або послідовний порт. Таким чином, перш ніж розпочати процес введення/виведення, у програмі має бути виконано відповідні настроєння (зокрема, швидкість приймання і передавання даних, довжина інформаційного посилання, наявність або відсутність біта контролю парності, наявність або відсутність режиму переривань, тощо).

Уведення/виведення символів з допомогою функцій *getchar()* і *putchar()*

Найпростішими функціями введення/виведення в мові C є `getchar()`

і `putchar()`, які призначено для посимвольного обміну даними (обидві оголошено в стандартному заголовному файлі `stdio.h`). Функція `getchar()` повертає символ, прийнятий від *UART/USART* або за послідовним інтерфейсом, а функція `putchar()`, навпаки, виводить символ. Інші стандартні функції введення/виведення базуються на них.

Функції `getchar()` і `putchar()` у компіляторах для мікроконтролерів реалізують очікування готовності саме приймача *UART/USART* для передання/приймання символу. У тому випадку, коли необхідно організувати обмін даними за іншим інтерфейсом (наприклад, SPI), необхідно розробити власні функції з аналогічним змістом.

Розглянемо приклад простої програми, що виводить у нескінченному циклі через приймач *UART/USART* символи «1», «2», «3», «4» і «5». Вихідний код програми `putchar.c` для мікроконтролерів AVR наведено в лістингу 2.1 для пристрою AT90S8515.

Лістинг 2.1. Програма `putchar.c` для мікроконтролерів AVR:

```
#include <avr/io.h>
#include <stdio.h>
int main (void)
{
    const char str[] = "12345";
    int i;
    UBRR = 25; // Швидкість обміну через UART - 9600 бод
    UCR = 0x18; // Установлюємо розряди TXEN і RXEN для
    // активізації UART у режимі введення/виведення через
    // виводи 0 та 1 порту D
    // Розряд CHR9 = 0 - передання 8 бітів даних
    i = 0; // Індекс рядка символів
    while (1) // Нескінченний цикл
    {
        if (i > 4) i = 0; // Перевірка виходу за межі масиву
        putchar(str[i]); // Виводимо i-й символ через UART
    }
}
```

Директиви препроцесора

Як вже зазначалося вище, директиви препроцесора за походженням не належать до складу мови C, а використовуються лише для підстановки коду в текст програми. *Препроцесор* – це особлива програма, яка виконує попереднє оброблення даних на початку компіляції. Розглянемо стандартні директиви препроцесору.

Директива **#include**

У розділі «Структура програми мовою С» вже згадувалася найпоширеніша директива `#include`, яка використовується майже в усіх програмах для включення до тексту програми заголовних файлів з розширенням `.h`, або `.c` (які не містять функцію `main()`).

Ця директива може використовуватись у двох формах:

```
#include <ім'я_файлу>
або
#include " ім'я _файлу"
```

Якщо ім'я файлу знаходиться між знаками `< i >`, то файл вважається частиною стандартної бібліотеки, що поставляється разом із компілятором. Якщо ж ім'я файлу взято в подвійні лапки, то вважається, що цей файл розташовано в тій же папці, що й файл з вихідним кодом.

Директива **#define**

Препроцесор сприймає директиву `#define` як необхідність замінити в тексті програми певну послідовність символів іншою послідовністю. Формат директиви:

`#define` змінна послідовність фрагмента підстановки

Наприклад:

```
#define MyName "John Smith"
#define Condition (a > b)
#define Operation a = b
...
char str[] = MyName; // Рівнозначно char str[] = "John
Smith"; int a, b;
if Condition Operatrion; // Рівнозначно if (a > b) a = b;
```

У директиві `#define` можуть використовуватися параметри, завдяки чому вона стає дуже потужним і гнучким інструментом, що дає змогу замінювати один простий текстовий елемент складним виразом. Такі підстановки називають макросами.

Наприклад, вираз, у якому вибирається найбільше з двох значень, можна подати у вигляді такого макроса:

```
#define larger(x, y) ( (x)>(y) ?      (x) :      (y) )
```

Якщо визначено такий макрос, то код, який його використовує, може

мати вигляд

```
int a = 9;
int b = 7;
int c = 0;
c = larger(a, b);
```

який нагадує виклик функції, однак це не функція, оскільки компілятор отримає від препроцесора останній рядок у такому вигляді:

```
C = ( (a) > (b) ? (a) : (b) );
```

Основна відмінність макроса від функції полягає в тому, що код макропідстановки вставляється препроцесором у програмний код скрізь, де трапляється заданий текстовий елемент, код функції визначається тільки в одному місці, а в тих місцях, де зазначено її ім'я, здійснюється виклик цього коду.

Існує ще одна відмінність, яка є особливо важливою під час програмування мікроконтролерів: при використанні макросів, на відміну від функцій, нічого не міститься в стек, що дає змогу заощадити оперативну пам'ять.

Крім того, макроси перетворюються на звичайний код, який виконується швидше, ніж код функції, на виклик якого й повернення керування у викличну функцію витрачається додатковий машинний час. Нарешті, під час використання макросів не потрібно формального оголошення типів даних.

Перелічимо деякі правила використання директиви `#define`: при створенні коментарів у рядку з `#define` завжди використовується коментар вигляду `/*...*/`; слід пам'ятати, що кінець рядка – це кінець `#define`, і весь текст зліва слід замінити текстом праворуч; для перетворення параметра макроса на текстовий рядок можна вказати перед ним символ `"#"`, наприклад:

```
#define OutString(s) puts(#s) OutString(Line);
//Рівнозначно puts("Line");
```

Для конкатенації двох параметрів можна скористатися оператором `##`, наприклад:

```
#define Concat(x, y) x ## y
int i = Concat(2,1); //Рівнозначно int i =21
```

Для перенесення тексту підстановки на інший рядок використовується символ зворотної косої, наприклад:

```
#define LongStr  "0 1 2 3 4 5 6 7 8 9  10 \
11 12 13 14 15 16 17 18 19 20 "
```

Для скасування визначення використовується директива `#undef`, наприклад:

```
#define A_Char 'A'
...
#undef A_Char
#define A_Char 'a'.
```

Директиви умовної компіляції

Багато мікроконтролерів відрізняються лише деякими параметрами, кількістю виводів, місткістю пам'яті й розміщенням регістрів. Це дає змогу створювати мовою C програмний код для всього модельного ряду. Однак для цього слід якимось чином замінити параметри, що є різними для різних моделей. Для таких цілей використовуються директиви умовної компіляції `#ifdef`, `#ifndef`, `#else`, `#endif` та `#elif`.

Синтаксис для директиви `#ifdef`:

```
#ifdef ім'я_макроса
    Послідовність_операторів_1
#else
    Послідовність_операторів_2
#endif
```

Якщо ім'я макроса визначено в програмі, то компілюється перша послідовність операторів, інакше — друга (гілки `#else` може не бути).

Приклад використання:

```
#define DEBUGGING_ON
#ifdef DEBUGGING_ON
    #use rs232 (baud=9600, xmit=PIN_C6, rcv=PIN_C7)
#endif
```

Синтаксис для директиви `#ifndef`:

```
#ifndef ім'я_макроса
    Послідовність_операторів_1
#else
    Послідовність_операторів_2
```

```
#endif
```

На відміну від директиви `#ifdef` перша послідовність операторів виконується в разі, якщо ім'я макроса у програмі не визначено.

Для умовної компіляції можна також скористатися директивами `#if`, `#elif`. Їх синтаксис є таким:

```
#if вираз1
    Послідовність_операторів_1
#elif вираз2
    Послідовність_операторів_2
#else
    Послідовність_операторів_3
#endif
```

Ця конструкція працює аналогічно умовному оператору `if-else`. Компілятор оцінює вирази після `#if` і `#elif` доти, доки одне з них не дасть у результаті `TRUE`, після чого в текст програми підставляється відповідна послідовність операторів. Якщо обидва вирази дають `FALSE`, то підставляється послідовність операторів після директиви `#else` (якщо вона є).

Припустимо, для передання й приймання даних через UART в абстрактному мікроконтролері `SomeMicX` використовуються виводи 12 і 13, у мікроконтролері `SomeMic8` – виводи 6 і 14, а у `SomeMic4` – виводи 1 і 2. Тоді можна створити заголовний файл `SomeMic.h` і включити в нього такі директиви:

```
#if SomeMicX == 16
    #define TXD 12
    #define RXD 13
#elif SomeMicX == 8
    #define TXD 6
    #define RXD 14
#elif SomeMicX == 4
    #define TXD 1
    #define RXD 2
#else
    #error "Pins TXD и RXD for SomeMicX are not defined"
#endif
```

Тепер, якщо програмний проект буде побудовано на мікроконтролері `SomeMic8`, у файл заголовка слід помістити такий текст:

```
#ifndef SomeMicX
#define SomeMicX = 8
#include <SomeMic.h>
#endif
```

За директивою `#ifndef` препроцесор додасть до тексту програми рядок `#define SomeMicX = 8` і `#include <SomeMic.h>`. Оскільки після цього елемент `SomeMicX` набуде значення 8, то будь-яке повторне оброблення директиви `#ifndef` не призведе до дублювання у вихідному тексті відповідної інформації. Іншими словами, вміст заголовного файлу `SomeMic.h` буде поміщено у вихідний код файлів, які використовують заголовковий файл з `#ifndef`, лише один раз.

Використання директиви `#ifndef` з наступними директивами `#define` та `#include` — стандартний прийом, що дає змогу уникнути дублювання інформації із заголовних файлів у вихідному коді проекту. Якщо цього не зробити, то під час дублювання компілятор зазвичай видає безліч повідомлень про помилки, пов'язані з багаторазовим оголошенням змінних.

3. ЦИФРОВІ ПРИСТРОЇ АВТОМАТИКИ

Практична робота № 1

ВИВЧЕННЯ РОБОТИ БАЗОВИХ ЛОГІЧНИХ ЕЛЕМЕНТІВ СТРУКТУРИ ТТЛ

Мета роботи – набуття навичок задання логічних автоматів та аналізу роботи логічних пристроїв автоматики та аналізу логічних функцій (ЛФ).

Прилади й обладнання: ЕОМ з установленою САПР MAX+PLUS II (ALTERA) або MATLAB.

Завдання:

- 1) мінімізувати ЛФ і розробити цифровий автомат реалізації цієї ЛФ (за варіантом табл. 3.1);
- 2) проаналізувати дію цифрового автомата здійснивши його симуляцію в системі САПР MAX + PLUS II або MATLAB.

Теоретичні відомості

Електрична схема й принцип роботи базового елемента ІС ТТЛ К155, КМ 155

Базовим логічним елементом серії ТТЛ вважають логічний елемент структури І-НІ (рис. 3.1).

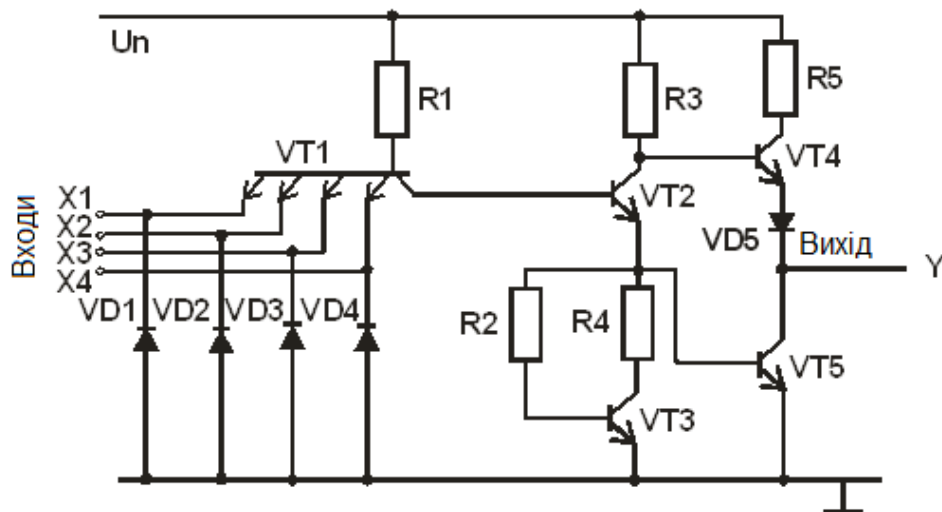


Рис. 3.1. Принципова електрична схема базового логічного елемента структури ТТЛ типу 4I-HI

Схема містить три основні каскади: вхідний на транзисторі VT1, що реалізує функцію 4I: фазорозподільний на транзисторі VT2; вихідний підсилювач на транзисторах VT4, VT5.

Вхідний каскад працює таким чином. Під час одночасного подання на всі входи напруги високого рівня (логічної 1) струм через резистор R1 пройде через перехід база – колектор транзистора VT1 у базу транзистора VT2; при цьому на колекторі транзистора VT1 рівень напруги значно підвищиться. Якщо на один із входів подається напруга низького рівня (логічний нуль), то струм через резистор R1 буде виходити зі схеми через перехід база – емітер транзистора VT1: на колекторі VT1 установиться низький рівень напруги. До всіх входів вхідного каскаду під'єднано демпферні діоди VD1 – VD4, які призначено для обмеження імпульсів напруги негативної полярності.

Фазорозподільний каскад виконано на транзисторі VT2, до емітера якого під'єднано коригувальну ланку R2, R4, VT3, що поліпшує передатну характеристику і завадостійкість схеми. Коли транзистор VT1 пропускає в базу транзистора VT2 керувальний струм, напруга на емітері транзистора VT2 може зростати тільки до значення U_{BE} транзистора VT5; коли транзистор VT1 не пропускає його через резистор R3 і ланку R2, R4, VT3 проходить лише тільки вихідний струм, тому напруга на емітері транзистора VT2 є близькою до нуля, а на колекторі – до напруги живлення. При паралельному з'єднанні двох або більше подібних фазорозподільних каскадів може бути реалізовано логічну функцію АБО.

Вихідний каскад містить транзистори VT4, VT5, діод VD5. Якщо хоча б на один із входів схеми подано логічний 0, то транзистори VT2, VT3 і VT5 будуть закритими. Через резистор R3 проходить струм розтікання транзистора VT2 і базовий ток транзистора VT4, тому напруга на колекторі

транзистора VT2 буде близькою до напруги живлення, а на виході схеми – нижчою за напругу колектора транзистора VT2 на величину $U_{VD5} + U_{BEVT4}$ (логічна 1).

Якщо на всі входи схеми подано потенціал логічну 1, транзистори VT2, VT3 і VT5 відкритими і вихідна напруга дорівнюватиме напрузі насиченого колекторно-емітерного переходу транзистора VT5.

До складу мікросхеми сім'ї ТТЛ належить також логічний елемент І-НІ без колекторного навантаження у вихідному каскаді. Це так званий елемент І-НЕ з відкритим колектором. Його призначено для роботи на зовнішнє навантаження, Як такий може бути використано електромагнітні реле, індикаторні прилади тощо.

Мікросхеми ТТЛ розраховано на напругу джерела живлення $5\text{ В} \pm 10\%$. Рівні логічних нуля й одиниці мають відрізнятися якомога більше. Різними є порогова напруга логічної одиниці $U_{пор1}$: найменша напруга високого рівня на вході мікросхеми, при якій напруга на виході змінюється від рівня логічного нуля до рівня логічної одиниці, а також порогова напруга логічного нуля $U_{0пор}$ – найбільша напруга низького рівня на вході мікросхеми, при якій напруга на виході змінюється від рівня логічної одиниці до рівня логічного нуля. Для мікросхем сім'ї ТТЛ $U_{1пор} = 2.4\text{ В}$, $U_{0пор} = 0.4\text{ В}$.

Порядок виконання роботи

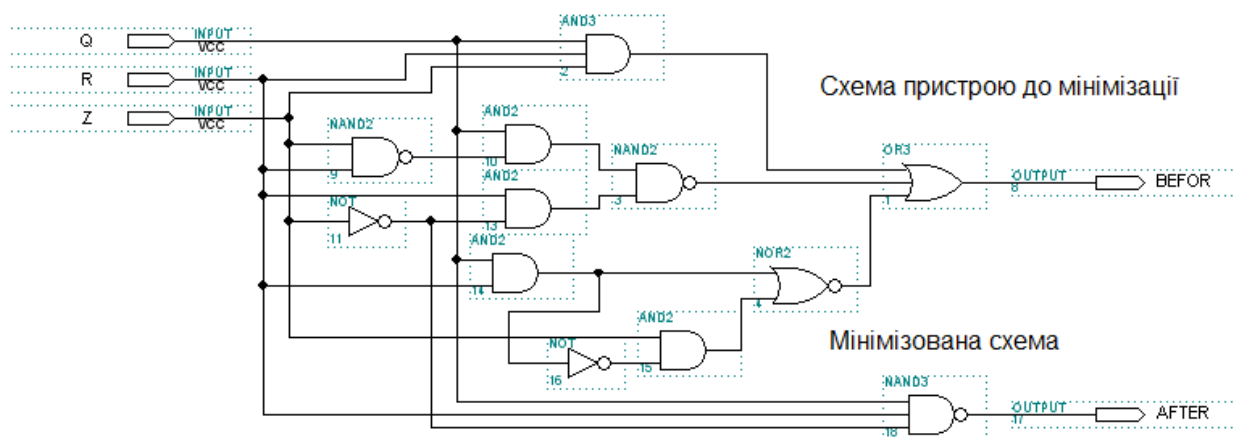
1. Мінімізація функції $F = QRZ + \overline{QRZRZ} + \overline{QR} + \overline{ZRQ}$:

$$QRZ + \overline{QRZRZ} = QRZ + !Q + RZ + !R + Z = !Q + !R + Z;$$

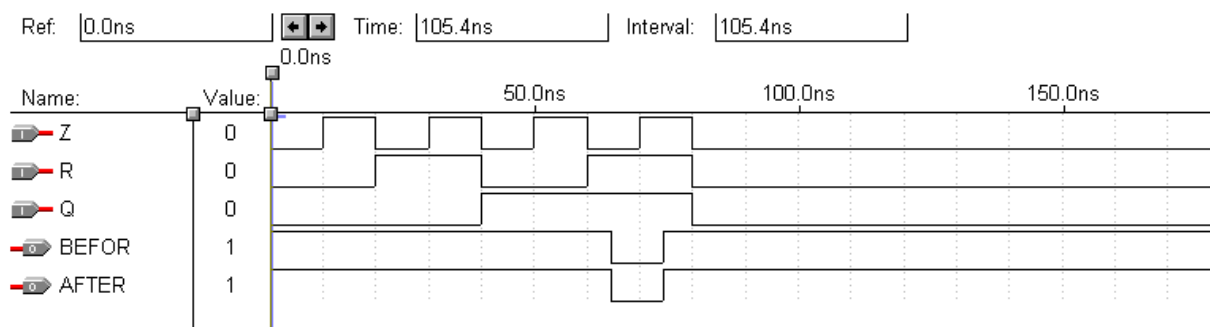
$$\overline{RQ} + \overline{ZRQ} = (!Q + !R)(!Z + RQ) = !Q!Z + 0 + !R!Z + 0;$$

$$F = !Q + !R + Z + !Q!Z + !R!Z = !Q + !R + Z = !(QR!Z).$$

2. Перевірка правильності мінімізації логічної функції здійснюється методом порівняння часових діаграм роботи логічного блока (ЛБ) до й після мінімізації його ЛФ: якщо діаграми збігаються, то роботу виконано правильно. Симуляцію роботи ЛБ проводимо в САПР, що здатна генерувати часові діаграми роботи логічного пристрою: MATLAB, MAX + PLUS II, WORKBench, OwenLogic, WisSim. Часові діаграми роботи ЛБ зображено на рис. 3.2. ЛБ являє собою дешифратор стану $!(QR!Z)$ ЛФ трьох логічних змінних.



а



Часові діаграми: BEFOR - до мінімізації
AFTER - після

б

Рис. 3.2. Схема ЛБ, задана ЛФ $QRZ + \overline{QRZRZ} + \overline{QR} + \overline{ZRQ}$: а – до й після мінімізації; б – часові діаграми роботи вихідного й мінімізованого ЛБ

Таблица 3.1

Номер варіанта	Вираз логічної функції
1	$\overline{UUV} + UUVW + \overline{U} \overline{VW}$
2	$\overline{AABAB} + \overline{BABAC} + \overline{BC} + \overline{ABC}$
3	$\overline{LK} + \overline{K} + \overline{LMK} + \overline{L} + \overline{MKM} + \overline{LLK}$
4	$\overline{QTR} + \overline{Q} + \overline{TR} \overline{RT} + \overline{TQ}$
5	$\overline{QRRZ} + \overline{QR} + \overline{RQRZ} + \overline{QR} \overline{Z}$
6	$\overline{AB} + \overline{CAB} + \overline{A} + \overline{BC} + \overline{BAC}$
7	$\overline{AABBC} + \overline{CAB} + \overline{B} \overline{CB} + \overline{BC} + \overline{AB}$
8	$\overline{ABC} + \overline{CDC} + \overline{ABC} + \overline{BCD} + \overline{DB} + \overline{C}$
9	$\overline{PQR} + \overline{PR} + \overline{PQ} + \overline{RRPQ}$
10	$\overline{QRZQRZ} + \overline{QRZRQ} + \overline{Z}$
11	$\overline{PTQR} + \overline{T} + \overline{QRT} + \overline{TRQ}$
12	$\overline{QRZ} + \overline{QQZ} + \overline{RZ} + \overline{QRRZ}$

Номер варіанта	Вираз логічної функції
13	$\overline{ML} + \overline{LK} + \overline{LM} + \overline{K} + \overline{KM} + \overline{LK}$
14	$\overline{AB} + \overline{ABCD} + \overline{CBC} + \overline{DC} + \overline{ACB} + \overline{BD} + \overline{BDC}$
15	$\overline{KL} + \overline{M} + \overline{K} + \overline{KLM} + \overline{MLM} + \overline{LM}$
16	$\overline{Q} \overline{R} + \overline{Z} + \overline{QRZ} + \overline{Q} + \overline{QRZ} + \overline{Z}$
17	$\overline{AB} + \overline{D} + \overline{C} + \overline{DC} + \overline{ACB} + \overline{BD} + \overline{BDC}$
18	$\overline{KL} + \overline{M} + \overline{K} + \overline{LM} + \overline{K} + \overline{LM}$
19	$\overline{Q} \overline{R} + \overline{Z} + \overline{QRZ} + \overline{Q} + \overline{R} + \overline{Z}$
20	$\overline{C} + \overline{DC} + \overline{AB} + \overline{BD} + \overline{BDC}$
21	$\overline{P} \overline{QPR} + \overline{R} + \overline{PQ} + \overline{R} + \overline{RQ}$
22	$\overline{AB} + \overline{CAB} + \overline{AB} + \overline{BC} + \overline{BA}$
23	$\overline{P} + \overline{QR} + \overline{T} + \overline{QR} + \overline{TR} + \overline{TQ}$
24	$\overline{ML} + \overline{K} + \overline{LKM} + \overline{K} + \overline{M} + \overline{LK}$
25	$\overline{Q} + \overline{RZ} + \overline{QR} + \overline{RZ} + \overline{QR} + \overline{RZ}$
26	$\overline{A} + \overline{BC} + \overline{AB} + \overline{CD} + \overline{AC}$
27	$\overline{L} + \overline{LN} + \overline{ML} + \overline{L} + \overline{ML} + \overline{MN} + \overline{NL}$
28	$\overline{QR} + \overline{Z} + \overline{QR} + \overline{R} + \overline{Z} + \overline{QR} + \overline{Z}$
29	$\overline{LKK} + \overline{LMK} + \overline{L} + \overline{MLK} + \overline{LLK}$
30	$\overline{AA} + \overline{AB} + \overline{BAB} + \overline{CA} + \overline{C} + \overline{A} + \overline{BC}$
31	$\overline{PQZZ} + \overline{PQ} + \overline{P} + \overline{Q} + \overline{PZQ}$
32	$\overline{QR} + \overline{Z} + \overline{QRZ} + \overline{QRZ}$
33	$\overline{ABC} + \overline{A} + \overline{B} + \overline{A} + \overline{A} + \overline{BC} + \overline{AB}$
34	$\overline{ABA} + \overline{ABC} + \overline{A} + \overline{AC} + \overline{BA}$
35	$\overline{QPQ} \overline{R} + \overline{Q} \overline{PR} + \overline{Q} + \overline{PR}$
36	$\overline{K} + \overline{MK} * \overline{LM} + \overline{KM} + \overline{KLM}$
37	$\overline{CAC} + \overline{A} + \overline{BCB} + \overline{AB} + \overline{C}$
38	$\overline{X} \overline{YZ} + \overline{XY} \overline{ZYY} + \overline{Z} + \overline{XZ}$
39	$\overline{ABC} + \overline{AB} + \overline{CD} + \overline{ADC}$
40	$\overline{AB} + \overline{CAB} + \overline{A} + \overline{BC} + \overline{BA}$

Контрольні запитання

1. У чому полягає суть означення «базовий логічний елемент»?
2. Сформулюйте схемотехнічні принципи побудови базового логічного елемента структури ТТЛ.
3. Наведіть параметри й характеристики логічних елементів:
 - параметри напруги й струму;
 - параметри часу;
 - параметри потужності та енергії;
 - параметри навантажувальної здатності та завадостійкості.
4. На чому ґрунтується висока технологічність складання й налаштування цифрових пристроїв автоматики?
5. Наведіть переваги й недоліки цифрової техніки порівняно з аналоговою.
6. Назвіть методи надання логічних функцій.
7. Назвіть методи проектування логічних блоків за наявності умов заданого логічного базису.
8. Поясніть принцип дії базового логічного елемента (ЛЕ) структури ТТЛ. Якими є переваги й недоліки базових ЛЕ?
9. Наведіть означення порогового рівня напруги логічного нуля й логічної одиниці.

Практична робота № 2

РОЗРОБЛЕННЯ ЦИФРОВОГО КЕРУВАЛЬНОГО АВТОМАТА

Мета роботи – набуття навичок розроблення мікропроцесорного керувального пристрою, аналізу й синтезу МП пристроїв автоматики.

Прилади й обладнання: лабораторний стенд з цифровими логічними пристроями, джерелом постійної напруги $+5\text{ V} \pm 10\%$, 1 A , ЕОМ з встановленою САПР MAX + PLUS II (ALTERA) або MATLAB (MathWorks), середовищем програмування MCS-51 або ARDUINO IDE (WinAvr).

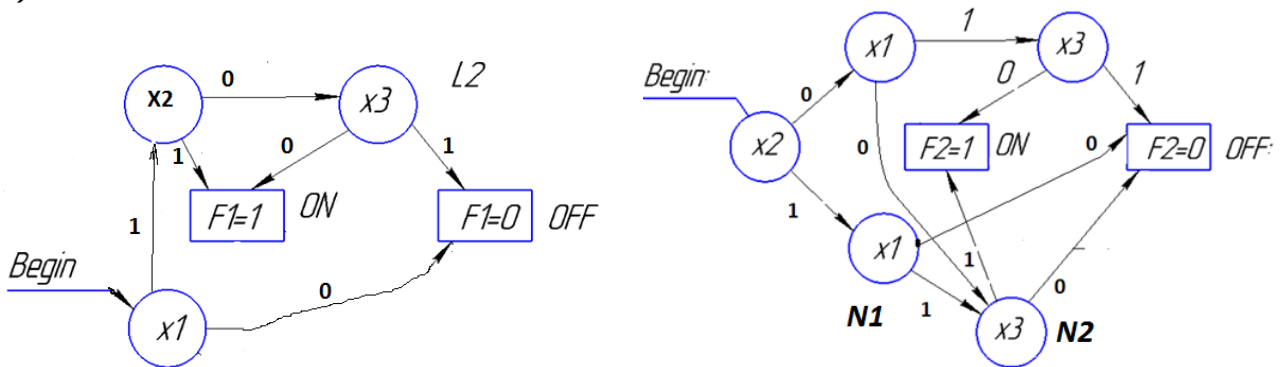
Завдання: розробити мікропроцесорний керувальний автомат (КА) реалізації двох логічних функцій (ЛФ) $F1$ і $F2$ трьох логічних змінних $X1$, $X2$ і $X3$.

Порядок виконання роботи

1. Записати логічні функції, задані таблично у вигляді аналітичного виразу: $F1 = x1x2x3 + x1!x2!x3 + x1x2!x3$;
 $!F2 = !x1!x2x3 + x1!x2!x3 + x1x2x3$.

2. Провести сумісну мінімізацію логічних функцій, використовуючи правила аналітичної мінімізації. Записати отримані мінімальні форми: $F1 = x1(x2 + !x3)$; $!F2 = !x2(x1 \oplus x3) + x1x2x3$.

3. Розробити графи бінарних алгоритмів ЛФ: $F1(x1, x2, x3)$ і $F2(x1, x2, x3)$:



4. Скориставшись графами ЛФ, розробити програмні реалізації ЛФ мовою асемблера EOM Intel8051 або мовою C у середовищі ARDUINO IDE (за умов дистанційного навчання).

5. Скласти звіт про виконання роботи, форму якого наведено нижче.

Звіт

1. Таблиця заданих логічних функцій $F1(x1, x2, x3)$ і $F2(x1, x2, x3)$ (табл. 3.2).

Таблиця 3.2

x_3	0	0	0	0	1	1	1	1
x_2	0	0	1	1	0	0	1	1
x_1	0	1	0	1	0	1	0	1
F_1	0	1	0	1	0	0	0	1
F_2	1	0	1	1	0	1	1	0

2. Запис довершеної диз'юнктивної нормальної форми (ДДНФ) двійкових логічних функцій $F1$ і $F2$ та їх сумісна мінімізація:

$$F1 = x_1 x_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 \bar{x}_3 = x_1 x_2 (x_3 + \bar{x}_3) + x_1 \bar{x}_2 \bar{x}_3 =$$

$$= x_1 x_2 + x_1 \bar{x}_2 \bar{x}_3 = x_1 (x_2 + \bar{x}_2 \bar{x}_3);$$

$$\bar{F}_2 = \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_3 = \bar{x}_2 (\bar{x}_1 x_3 + x_1 \bar{x}_3) + x_1 x_2 x_3 =$$

$$= \bar{x}_2 (x_1 \oplus x_3) + x_1 x_2 x_3,$$

де « $\oplus$ » – оператор складання за модулем 2.

3. На основі мінімальної форми ЛФ будуємо графи бінарних алгоритмів (рис. 3.3 і 3.4) і програму реалізації ЛФ мовою асемблера *Intel8051* [5]:

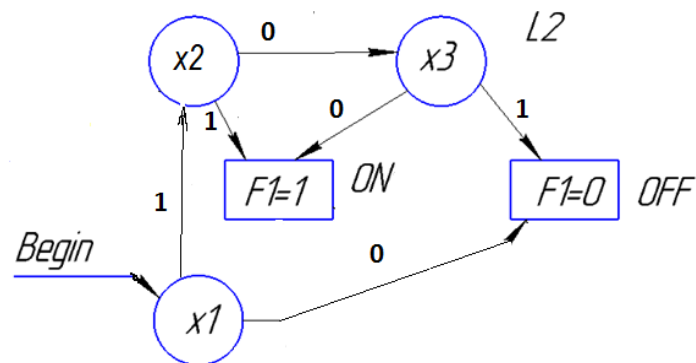


Рис. 3.3. Граф бінарного алгоритму реалізації ЛФ $F1(x1, x2, x3)$

Реалізація бінарного алгоритму $F1$ мовою асемблера *Intel8051* має вигляд

```

BeginF1:
    JNB    X1,  OFF
    JB     X2,  ON
           JB     X3,  OFF
ON:
    SETB   F1
    RET
OFF:
    CLR    F1
    RET

```

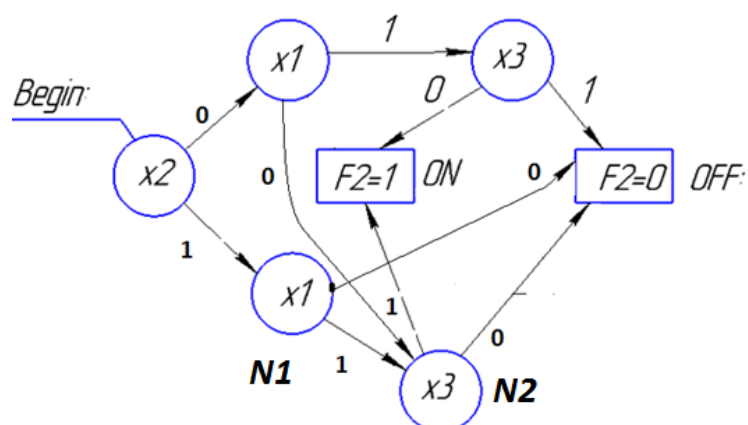


Рис. 3.4. Граф бінарного алгоритму реалізації ЛФ $F2(x1, x2, x3)$

Реалізація бінарного алгоритму $F2$ мовою асемблера *Intel8051* має вигляд

```

BeginF2:
    JB    X2, N1      ; Jmp N1 if Bit X2 set
    JNB   X1, N2      ; Jmp N2 if Bit X1 Not set
    JB    X3, OFF_F2
ON_F2:
    CLR   F2          ; з урахуванням інверсії F2
    RET
N1: JNB   X1, OFF
N2: JB    X3, ON_F2
OFF_F2:
    SETB  F2          ; з урахуванням інверсії F2
    RET

```

За умов дистанційного навчання розроблення керувальної програми (КП) проводимо з використанням середовища *ARDUINO IDE* з наступним налагодженням на ресурсі *Wokwi.com* (*Tinkercad.com*) або в середовищі *WinAvr*.

Висновки

1. Програмування МП автомату складається з певних етапів:
 - а) зведення логічних функцій до їх канонічних форм;
 - б) сумісна мінімізація логічних функцій включення, блокування, захисту, оброблення аварій, індикації тощо;
 - в) графічне подання алгоритмів, зазначених вище, з метою найпростішого переходу до програмного подання будь-якою алгоритмічною мовою;
 - г) безпосереднє відтворення програмного коду алгоритмів.
2. Тестування програм залежно від умов експлуатації відбувається на макеті устаткування, який теж може бути створено програмно в тому ж середовищі програмування й додано до програмного забезпечення. Установлюється відповідність вимогам поставленого технічного завдання, виявлення помилок та їх усунення.

У звіті про виконання роботи необхідно навести перелік використаних літературних джерел.

Додатком до звіту про виконання роботи є лістинг трансляції з мови асемблера (MCS-51) або лістинг програми мовою C (*ARDUINO IDE* або *WinAvr*).

Лістинг трансляції КП мовою асемблера INTEL8051 у середовищі MCS-51

```

MCS-51 MACRO ASSEMBLER  SHEVA
01/27/22;    PAGE      1

```

```

DOS 5.0 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN SHEVA.OBJ
ASSEMBLER INVOKED BY:  U:\202\ASM51\ASM51.EXE SHEVA.ASM

```

LOC	OBJ	LINE	SOURCE
		1	
0090		2	X1 BIT P1.0; PIN12 INPUT
0091		3	X2 BIT P1.1; PIN13 INPUT
0092		4	X3 BIT P1.2; PIN14 INPUT
00B0		5	F1 BIT P3.0; PIN2 OUTPUT
00B1		6	F2 BIT P3.1; PIN3 OUTPUT
		7	
0000		8	ORG RESET
0000 020037		9	JMP MAIN
		10	BEGIN_F1:
0003 209109		11	JB X2,L1
0006 30900F		12	JB X1,ON
0009 20920C		13	JB X3,ON
000C C2B0		14	OFF_F1: CLR F1
000E 22		15	RET
000F 309003		16	L1: JNB X1,L3
0012 209203		17	JB X3,L3
0015 2092F4		18	L3: JB X3,OFF_F1
0018 D2B0		19	ON_F1: SETB F1
001A 22		20	RET
		21	
		22	BEGIN_F2:
0018 309106		23	JNB X2,N1
001E 20900C		24	JB X1,N2
0021 D2B1		25	ON_F2: SETB F2
0023 22		26	RET
0024 20900B		27	N1: JB X1,N3
0027 3092F7		28	JNB X3,ON_F2
002A C2B1		29	OFF_F2: CLR F2
002C 22		30	RET
002D 2092DC		31	N2: JB X3,OFF_F2
0030 309207		32	JMP ON_F2
0032 309207		33	N3: JNB X3,OFF_F2
0035 B0E1		34	JMP ON_F2
		35	
0037 1103		36	MAIN: CALL BEGIN_F1
0039 111B		37	CALL BEGIN_F2
003B 80FA		38	JMP MAIN
		39	END

MCS-51 MACRO ASSEMBLER SHEVA
PAGE 2

01/27/22;

SYMBOL TABLE LISTING

N A M E	T Y P E	V A L U E	A T T R I B U T E S
BEGIN. . .	C ADDR	0003H	A
F1	B ADDR	00B0H.0	A
F2	B ADDR	00B0H.1	A
L1	C ADDR	000CH	A

```

MAIN . . . C ADDR 0026H A
N1 . . . C ADDR 001BH A
N2 . . . C ADDR 001EH A
OFF. . . C ADDR 0009H A
OFF2 . . . C ADDR 0018H A
ON . . . C ADDR 000FH A
ON2. . . C ADDR 0023H A
P1 . . . D ADDR 0090H A
P3 . . . D ADDR 00B0H A
RESET. . . C ADDR 0000H A
START. . . C ADDR 0012H A
X1 . . . B ADDR 0090H.0 A
X2 . . . B ADDR 0090H.1 A
X3 . . . B ADDR 0090H.2 A
REGISTER BANK(S) USED: 0
ASSEMBLY COMPLETE, NO ERRORS FOUND

```

Реалізація ПФ F1(x1, x2, x3) мовою C у середовищі ARDUINO IDE:

```

const int ledPin = 13;           // the number of the LED pin
const int buttonPin_1 = 2;       // the number of the pushbutton pin X1
const int buttonPin_2 = 3;       // the number of the pushbutton pin X2
const int buttonPin_3 = 4;       // the number of the pushbutton pin X3
// variable
    int: x1, x2, x3;
    bool: F1=FALSE;

void setup() {
    pinMode(ledPin, OUTPUT); pinMode(buttonPin_1, INPUT);
    pinMode(buttonPin_2, INPUT); pinMode(buttonPin_3, INPUT);
    Serial.begin(9600); // Open serial communications and wait for port to
open.
    while (!Serial) { // Wait for serial port to open.}
Serial.println("to work redy");
    }
void loop() { // Незкінченний цикл
    x1 = digitalRead(buttonPin_1);
    x2 = digitalRead(buttonPin_2);
    x3 = digitalRead(buttonPin_3);
    if (x1=TRUE){
        if ((x2=TRUE) OR (x3=FALSE))
            { F1=TRUE; digitalWrite (ledPin, HIGH); }
            else { F1=FALSE; digitalWrite (ledPin,
LOW); }
    }
    Serial.println("Power_", F1);
}

```

Варіанти завдань

Розробити комбінаційну логічну схему (КЛС) функцій F1 і F2 трьох логічних змінних, заданих таблицею істинності, розробити релейно-контактні схеми й програмні реалізації цих алгоритмів мовою асемблера *Intel-8051* або C (*ARDUINO IDE*, *WINAVR*):

Варіант 1

X1	0	0	0	0	1	1	1	1
X2	0	x	1	x	0	0	1	1
X3	0	1	0	1	0	1	0	1
F1	0	0*	0	0*	0	1	1	1
F2	1	0	0	1	1	1	0	0

Варіант 2

X1	0	0	0	1	1	1	1	1
X2	0	0	1	1	x	0	x	1
X3	0	1	0	1	0	1	0	1
F1	0	1	0	0	1	1	1	1
F2	1	0	0	1	0*	1	0*	0

Варіант 3

X1	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X3	0	1	x	x	0	1	0	1
F1	0	1	0*	0*	0	0	1	1
F2	1	0	1*	1*	1	0	0	0

Варіант 4

X1	0	x	0	0	1	x	1	1
X2	0	0	1	1	0	0	1	1
X3	0	1	0	1	0	1	0	1
F1	0	0*	1	1	0	0*	1	0
F2	1	0	0	0	1	1	0	1

Варіант 5

X3	0	0	0	0	1	1	1	1
X2	0	x	1	x	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	0	0	1	1	1	0	1
F2	0	1*	1	1*	1	1	0	0

Варіант 6

X3	1	1	1	1	0	0	0	0
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	x	x	0	1
F1	0	0	0	1	0*	0*	0	1
F2	0	0	1	0	1*	1*	0	0

Варіант 7

X3	0	0	0	0	1	1	1	1
X2	1	1	0	0	1	1	0	0
X1	1	0	1	0	1	0	x	x
F1	0	0	0	1	1	1	0	1
F2	0	0	1	0	1	0	0*	0*

Варіант 8

X3	0	0	0	0	1	1	1	1
X2	x	0	x	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1*	0	1*	1	1	0	1	0
F2	0	1	1	0	1	1	0	0

Варіант 9

X3	0	0	0	0	1	1	1	1
X2	0	x	1	x	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1	1*	0	1*	0	1	0	1
F2	1	0*	1	0*	1*	1	0	0

Варіант 10

X3	0	0	0	0	1	1	1	1
X2	x	0	x	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1*	0	1*	1	1	0	0	1
F2	0	0	1	0	1	1	1	0

Варіант 11

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	x	x	0	1	0	1
F1	0	1	0*	0*	0	1	0	1
F2	0	1	1	0	0	1	1	0

Варіант 12

X3	x	1	1	1	x	0	0	0
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0*	1	1	1	0*	0	0	1
F2	0	1	1	0	0	1	1	0

Варіант 13

X3	0	0	0	0	1	1	1	1
X2	1	1	0	0	1	1	0	0
X1	0	1	x	x	0	1	0	1
F1	0	1	1*	1*	0	0	0	1
F2	0	1	1	0	0	1	1	0

Варіант 14

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	1	0	1	0	x	x	1	0
F1	0	1	1	1	0	1	0	1
F2	0	1	0	0	0*	0*	1	0

Варіант 15

X3	1	1	x	1	0	0	x	0
X2	1	1	0	0	1	1	0	0
X1	0	1	0	1	0	1	0	1
F1	0	1	0*	1	0	0	0*	1
F2	0	1	1*	1	0	0	1*	0

Варіант 16

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1	1	1	0	0	0	0	1
F2	0	1	1	0	0	1	1	0

Варіант 17

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1	1	1	0	0	0	1
F2	0	1	1	0	1	0	1	0

Варіант 18

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1	0	1	0	1	0	1
F2	1	0	1	0	0	1	1	0

Варіант 19

X3	0	x	0	0	1	x	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1*	1	1	0	1*	0	1
F2	0	1	1	0	0	0	0	1

Варіант 20

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	x	0	x	1
X1	0	1	0	1	0	1	0	1
F1	0	1	1	0	1	0	0	1
F2	0	1	1	0	0*	1	0*	0

Варіант 21

X3	0	0	0	x	1	1	1	x
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1	1	0	1*	0	1	0	1*
F2	0	1	0	0	0	1	1	1

Варіант 22

X3	1	x	1	1	0	x	0	0
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1	1*	0	1	0	1*	0	0
F2	0	0*	1	0	0	0*	1	1

Варіант 23

X3	0	0	0	0	1	1	1	1
X2	1	1	0	0	1	1	0	0
X1	1	0	x	x	1	0	1	0
F1	1	0	1*	1*	0	1	0	0
F2	0	1	1	0	0	1	1	1

Варіант 24

X3	0	0	0	0	1	1	1	1
X2	0	x	1	x	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1*	0	1*	0	1	0	0
F2	1	1	0	0	0	0	1	1

Варіант 25

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	x	x	0	1
F1	1	0	1	1	1*	1*	0	0
F2	0	1	0	1	0	0	1	1

Варіант 26

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	x	x	0	1	0	1	0	1
F1	0	1	1	0	0	0	0	1
F2	0*	0*	0	1	0	1	1	0

Варіант 27

X3	0	0	0	0	1	1	1	1
X2	0	x	1	x	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1	0*	1	0*	0	1	0	1
F2	0	1	0	0	1	1	0	1

Варіант 28

X3	0	0	x	0	1	1	x	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	0	1	1	0	1	0	0
F2	1	1	0*	0	0	1	0*	0

Варіант 29

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	x	x	0	1
F1	1	1	0	0	1*	1*	0	0
F2	0	1	1	0	1	0	1	0

Варіант 30

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1	0	0	0	1	1	0	0
F2	0	1	0	1	0	1	1	0

Варіант 31

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1	0	0	1	1	0	1
F2	1	0	1	1	1	1	0	0

Варіант 32

X3	1	1	1	1	0	0	0	0
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1	0	0	1	1	0	1
F2	1	0	1	1	1	1	0	0

Варіант 33

X3	0	0	0	0	1	1	1	1
X2	1	1	0	0	1	1	0	0
X1	0	1	0	1	0	1	0	1
F1	0	1	0	0	1	1	0	1
F2	1	0	1	1	1	1	0	0

Варіант 34

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	1	0	1	0	1	0	1	0
F1	0	1	0	0	1	1	0	1
F2	1	0	1	0	1	1	0	0

Варіант 35

X3	1	1	1	1	0	0	0	0
X2	0	0	1	1	0	0	1	1
X1	1	0	1	0	1	0	1	0
F1	0	1	0	0	1	1	0	1
F2	1	0	1	0	1	1	0	0

Варіант 36

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1	1	0	0	0	0	1
F2	1	0	0	1	0	1	0	0

Варіант 37

X3	x	0	0	0	x	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1*	0	0	1	1*	1	0	0
F2	1*	0	1	0	1*	0	0	0

Варіант 38

X3	0	0	0	0	1	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0	1	0	0	1	0	1	1
F2	0	0	1	0	1	1	0	1

Варіант 39

X3	0	0	0	0	1	1	1	1
X2	x	0	x	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	1*	0	1*	0	1	1	0	1
F2	0	1	1	1	1	1	0	0

Варіант 40

X3	x	0	0	0	x	1	1	1
X2	0	0	1	1	0	0	1	1
X1	0	1	0	1	0	1	0	1
F1	0*	1	1	0	0*	1	0	1
F2	1	0	1	0	0	1	0	0

X – довільне значення логічної змінної

Практична робота № 3

АНАЛОГО-ЦИФРОВЕ ПЕРЕТВОРЕННЯ ДАНИХ

Мета роботи: вивчення методів цифроаналогового (ЦА) та аналого-цифрового (АЦ) перетворення даних.

Прилади й обладнання: МП платформа *ARDUINO UNO*, базова ЕОМ.

Завдання: вивчити характеристики ЦАП і АЦП даних, розробити алгоритми й керувальні програми підтримки АЦП даних, скласти й відлагодити керувальні програми.

Теоретичні відомості

Параметри й характеристики АЦП

Існують загальні означення, які використовують під час роботи з аналого-цифровими перетворювачами.

До основних характеристик АЦП належать:

- розрядність перетворювача;
- час перетворення;
- діапазон вхідних напруг (струмів);
- помилки перетворення;
- лінійність передавальної характеристики;
- метод перетворення;
- шумові характеристики АЦП.

Правильний вибір оптимального поєднання характеристик АЦП для конкретного застосування потребує точного розуміння даних, що наводяться в технічній документації.

Так, під поняттям «роздільна здатність» АЦП розуміється задання максимальної помітної кількості рівнів вхідної величини, а «точність» характеризується сумарним відхиленням результату перетворення від дійсного значення.

Роздільну здатність АЦП вимірюють одиницею *LSB* (*Low Significant Bit*): $1\text{LSB} = U_{\text{ref}} / 2^N$, де N – число розрядів перетворювача, U_{ref} – опорна (зразкова) напруга. Точність перетворювача виражають числом, кратним *LSB*.

Зважаючи на вплив завад і недосконалість технології виготовлення АЦ перетворення код на його виході означає вимірювану величину з деякою похибкою. У специфікації АЦП наводять похибки, які вносить сам перетворювач. Ці похибки зазвичай поділяють на статичні й динамічні.

Статична похибка

У більшості випадків АЦП використовують для вимірювання сигналів, що повільно змінюються (наприклад, від датчиків температури, тиску, тензодатчиків тощо). При таких вимірах найважливішою є статична похибка вимірювання.

У специфікації на АЦП цей тип похибки визначається як сума адитивної (Offset) й мультиплікативної (Full-Scale) похибок, диференціальної (DNL) та інтегральної (INL) нелінійності, а також похибки квантування. Їх сукупність дає змогу повністю описати статичну похибку АЦП.

Передатна характеристика АЦП

Передатна характеристика АЦП – це функція змінення коду на його виході залежно від напруги на його вході. Графічно це шматково-лінійна функція з 2^N «ступенів», де N – розрядність АЦП. Кожен горизонтальний відрізок цієї функції відповідає одному зі значень вихідного коду АЦП.

Якщо з'єднати лініями початок цих горизонтальних відрізків (на межах переходу від одного значення коду до іншого), то ідеальна передатна характеристика являтиме пряму лінію, що проходить через початок координат.

На рис. 3.5 показано ідеальну передатну характеристику трирозрядного АЦП з контрольними точками на межах переходу коду. Вихідний код набуває найменшого значення (000b) при величині вхідного сигналу від 0 до $1/8$ повної шкали значення коду цього АЦП. Слід зазначити, що АЦП набуває повного значення коду (111b) при $7/8$ повної шкали вхідної напруги.

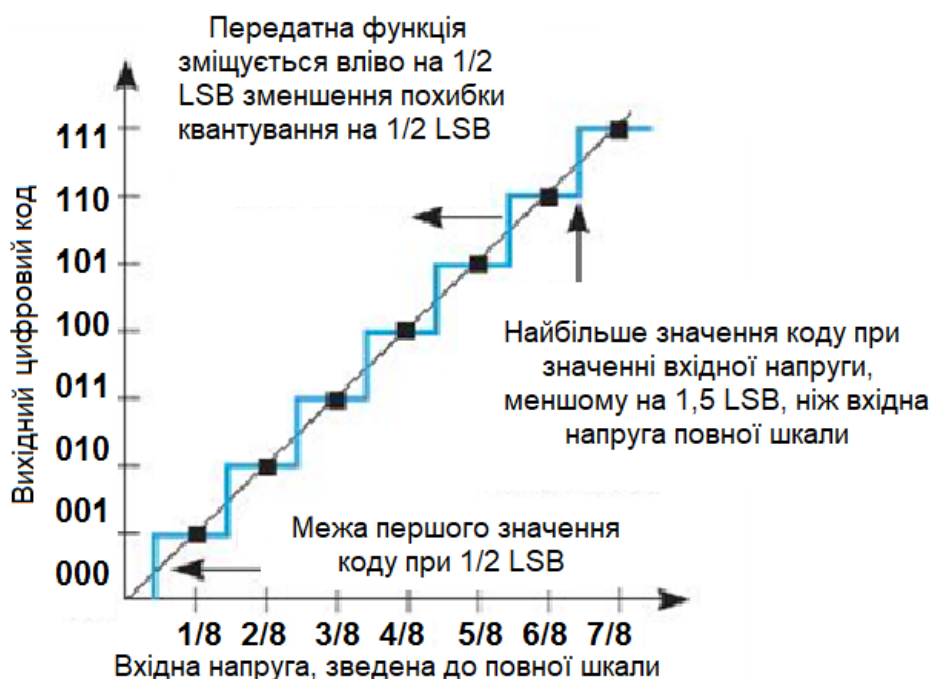


Рис. 3.5. Ідеальна передатна характеристика трирозрядного АЦП

Таким чином, максимального значення коду на виході АЦ перетворення набуває не при напрузі повної шкали, а при напрузі, меншій на 1LSB.

Передатну характеристику можна реалізувати зі зміщенням $-1/2$ LSB. Це досягається її зсувом уліво, що приводить до зміщення похибки квантування з діапазону $-1...0$ LSB у діапазон $-1/2...+1/2$ LSB (рис. 3.6).

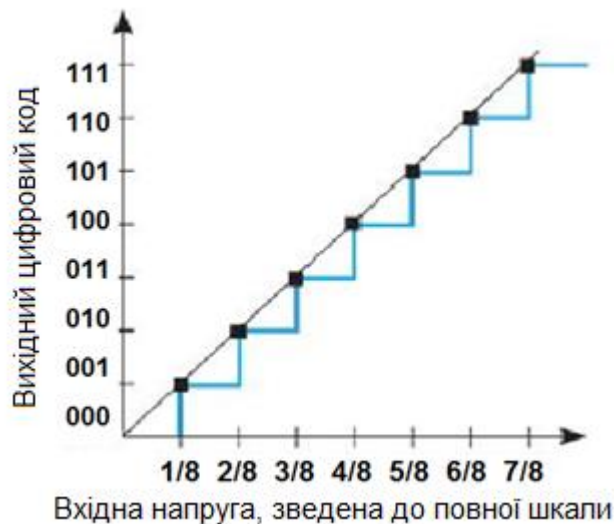


Рис. 3.6. Передатна характеристика трирозрядного АЦП зі зміщенням на $-1/2$ LSB

Унаслідок технологічного розкиду параметрів під час виготовлення інтегральних мікросхем реальні АЦ перетворення не мають ідеальної передатної характеристики. Відхилення від неї визначають статичну похибку АЦП, їх наводять у технічній документації.

Адитивну похибку (Offset Error) – похибку зміщення – можна визначити як зміщення всієї передатної характеристики ліворуч або праворуч відносно осі вхідної напруги.

Таким чином, у характеристику адитивної похибки АЦП навмисно введено зсув $1/2$ LSB.

Мультиплікативна похибка (Full-Scale Error) – похибка повної шкали – являє собою різницю між ідеальною та реальною передатними характеристиками у точці максимального вихідного значення за умови нульової адитивної похибки (коли немає зміщення). Це виявляється у зміні нахилу передатної функції, що показано на рис. 3.7.

Величина найменшого значущого розряду АЦП становить $U_{\text{ref}} / 2^N$, де U_{ref} – опорна напруга, N – кількість розрядів АЦП. Різниця напруг при кожному

кодівому переході має дорівнювати величині LSB. Відхилення від LSB, виражене у відсотках відносно нього, визначається як диференціальна нелінійність (DNL). Різниця між ідеальним і реальним інтервалами коду є похибкою диференціальної нелінійності.

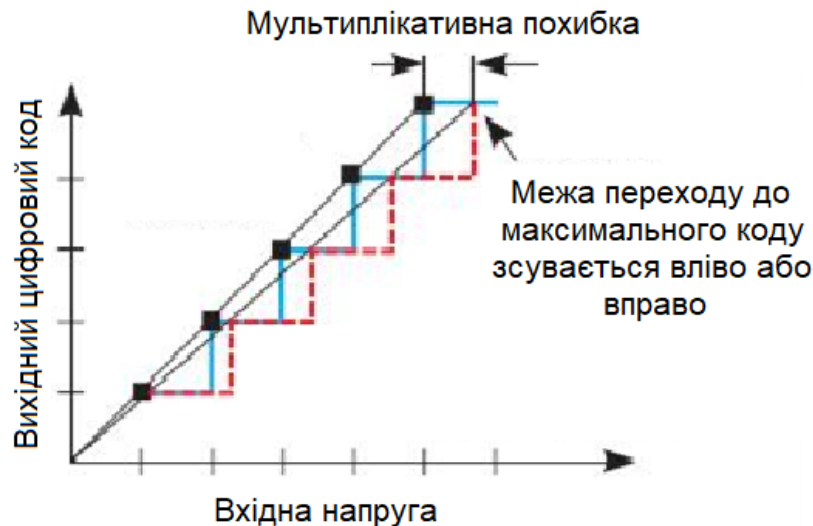


Рис. 3.7. Графічне відображення мультиплікативної похибки:
 - - - - передатна характеристика з мультиплікативною похибкою;
 ——— передатна характеристика без мультиплікативної похибки

На рис. 3.8 похибку DNL зображено у вигляді нерівних проміжків між сусідніми рівнями коду або як «розмитість» меж між переходами на передатній характеристиці АЦ перетворення.

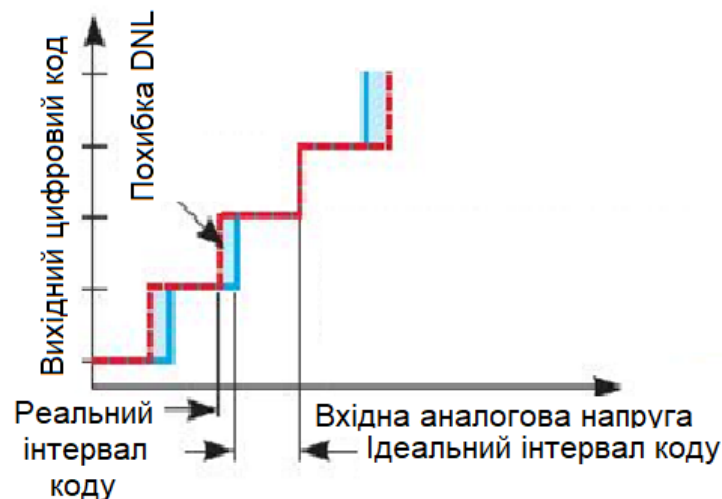


Рис. 3.8. Графічне зображення диференціальної нелінійності:
 - - - - передатна характеристика з похибкою DNL;
 ——— передатна характеристика без похибки DNL

Інтегральна нелінійність (INL) – це похибка, яка визначається відхиленням лінійної функції передатної характеристики АЦП від прямої лінії, як показано на рис. 3.9. Зазвичай передатна функція з інтегральною нелінійністю апроксимується прямою лінією методом найменших квадратів. Часто апроксимувальною прямою з'єднують найменше й найбільше значення. Інтегральну нелінійність визначають шляхом порівняння напруг, при яких відбуваються кодові переходи.

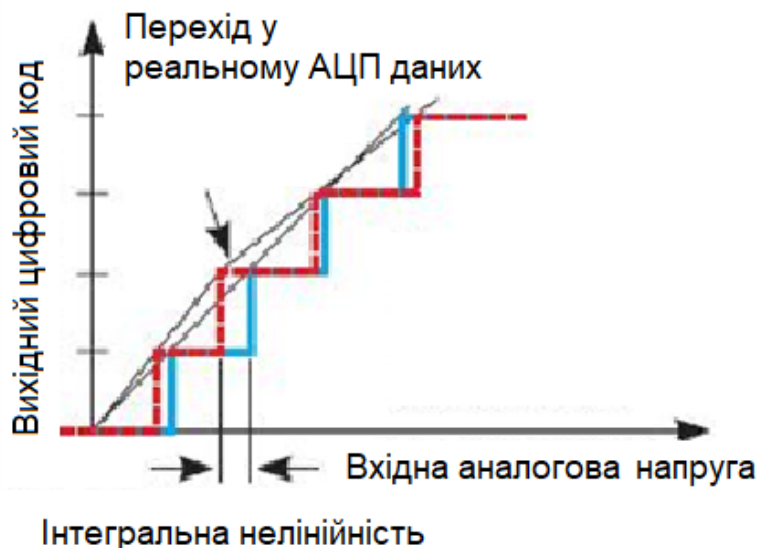


Рис. 3.9. Графічне зображення інтегральної нелінійності:

- - - - передатна характеристика з похибкою INL;
- передатна характеристика без похибки INL

В ідеальному АЦП переходи відбуватимуться при значеннях вхідної напруги, що точно відповідають величині LSB. У реальному перетворювачі така умова може виконуватися з похибкою. Різниця між «ідеальними» рівнями напруги, у яких відбувається кодовий перехід, і його реальними значеннями вимірюється в одиницях LSB і має назву інтегральної нелінійності. INL не пов'язана з DNL.

Похибка квантування – одна з найбільш значущих складових помилок АЦ перетворення – є результатом самого процесу перетворення, це похибка, спричинена значенням $U_{\text{ref}} / 2^N$ кроку квантування й визначається як $1/2$ величини LSB. Її не можна виключити при аналого-цифрових перетвореннях, оскільки вона є невід'ємною частиною цього процесу, характеризується роздільною здатністю АЦП і є постійною для АЦП з однаковою роздільною здатністю.

Динамічні характеристики АЦП зазвичай визначають з допомогою спектрального аналізу за результатами швидкого перетворення Фур'є

(ШПФ) над масивом вихідних значень АЦП, які відповідають деякому вхідному тестовому сигналу.

На рис. 3.10 показано приклад частотного спектра вимірюваного сигналу. Нульова гармоніка відповідає основній частоті вхідного сигналу. Усі інші являють собою шум, який містить гармонічні спотворення, тепловий шум, шум вигляду $1/f$ і шум квантування. Деякі складові можуть надходити на вхід АЦП із зовнішніх ланцюгів. Гармонічні спотворення, наприклад, можуть бути у вимірюваному сигналі й одночасно генеруватися АЦП під час перетворення.

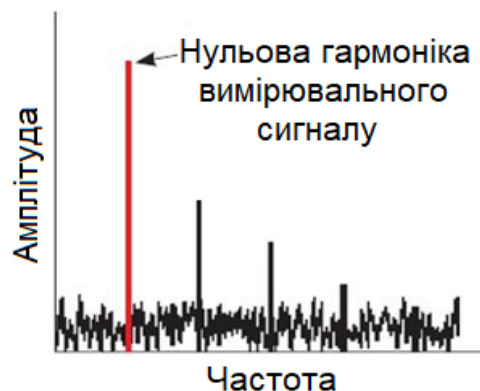


Рис. 3.10. Результати виконання ШПФ над вихідними даними АЦП

Відношення сигнал/шум (Signal/Noise Response) (рис. 3.11) – це відношення середньо-квадратичної величини вхідного сигналу до середньоквадратичного значення шуму (*Noise*) (за винятком гармонічних спотворень), виражене в децибелах:

$$SNR(dB) = 20 \log [V_{signal}(rms)/V_{noise}(rms)].$$

Це значення дає змогу вирахувати частку шуму в сигналі, що вимірюється відносно корисного сигналу.

Шум, що визначається під час розрахунку SNR, не містить гармонічних спотворень, але містить шум квантування. В АЦП з певною роздільною здатністю саме шум квантування обмежує можливості перетворювача теоретично кращим значенням відношення сигнал/шум:

$$SNR(db) = 6,02 N + 1,76,$$

де N – кількість розрядів АЦП.

Спектр шуму квантування АЦП стандартних архітектур має рівномірний розподіл за частотою. Тому величину цього шуму не можна зменшити шляхом збільшення часу перетворення й подальшого усереднення результатів. Шум квантування можна зменшити лише шляхом проведення вимірювань з допомогою АЦП більшої розрядності.

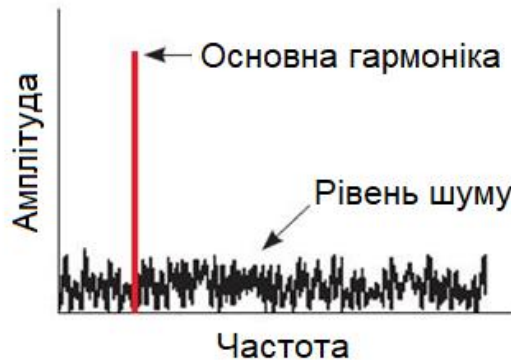


Рис. 3.11. Співвідношення сигнал/шум (SNR)

Особливість сигма-дельта АЦП полягає в тому, що спектр шуму квантування розподілено за частотою нерівномірно – він є зміщеним у бік високих частот. Тому, збільшивши час вимірювання (і відповідно кількість вибірок вимірюваного сигналу), нагромадивши й потім усереднивши отриману вибірку (фільтр нижніх частот), можна отримати результат вимірювань з більшою точністю. Звісно, при цьому загальний час перетворення збільшуватиметься.

Загальні гармонічні спотворення

Нелінійність у результатах перетворення даних призводить до виникнення гармонічних спотворень, які можна спостерігати як «викиди» у спектрі частот на парних і непарних гармоніках вимірюваного сигналу.

Ці спотворення визначають як загальні гармонічні спотворення (THD):

$$THD = 20 \log \left[\frac{\sqrt{V_2^2 + V_3^2 + \dots + V_n^2}}{V_1} \right].$$

Величина гармонічних спотворень зменшується на високих частотах до точки, у якій амплітуда гармонік стає меншою за рівень шуму. Таким чином, якщо аналізуємо вплив гармонічних спотворень на помилки перетворення, то це можна робити в усьому спектрі частот, обмежуючи амплітуду гармонік рівнем шуму або обмежуючи смугу частот для аналізу.

Наприклад, якщо в системі встановлено фільтр нижніх частот (ФНЧ), то верхні частоти не впливають на результат вимірювань і не враховуються.

Перетворення аналогового сигналу від датчика на цифровий код, який приймається й обробляється в МП, можна виконати декількома способами:

1) апаратним на основі БІС АЦ перетворення, що під'єднується до порту МП (наприклад, K1113ПВ1, K572ПВ1, AD7715, ADS1286, MAX1241 тощо). МК лише ініціалізує АЦП і після закінчення перетворення зчитує з нього готовий цифровий код. При застосуванні цього способу передбачається використання покупних БІС АЦП, параметри яких визначаються паспортними даними й гарантіями фірм-виробників. Перевагою такого способу є висока надійність БІС, недоліком – висока вартість, яка не завжди є виправданою;

2) апаратно-програмним на основі БІС ЦАП і програми зважування бітів (метод послідовних наближень). Цей спосіб характеризується достатньою швидкістю (десятки мікросекунд) і потребує використання досить простих і дешевих мікросхем ЦАП і компаратора. Метод є чутливим до різноманітних імпульсних перешкод (шумів процесора, імпульсних стрибків напруги живлення тощо) і за своєю суттю є диференціальним;

3) програмно-апаратним на основі методу подвійного інтегрування. Цей більш повільний спосіб (десятки й сотні мілісекунд) може забезпечити досягнення досить високої точності перетворень (до 18 розрядів). З додаткового обладнання необхідно мати два операційні підсилювачі й аналоговий мультиплексор з двома входами; точність досягається інтегровальною суттю методу перетворення й відносно великим часом перетворення;

4) інтегровальним – на основі використання перетворювача напруга/частота й програми вимірювання частоти сигналу. Це – найповільніший (секунди й десятки секунд) і найточніший інтегровальний метод перетворення;

5) сигма-дельта перетворювач на основі методу врівноваження заряду – найточніший і найдорожчий із середніх за швидкістю методів перетворення (24 двійкових розряди).

Метод послідовних наближень

З додаткової апаратури МП-системі використовують інтегральний цифроаналоговий перетворювач (ЦАП) на задану кількість входів (у прикладі їх вісім) і операційний порівнювальний підсилювач (компаратор).

На виході компаратора формується сигнал 0, якщо $U_{\text{ЦАП}} < U_{\text{вх}}$, і сигнал 1, якщо $U_{\text{ЦАП}} > U_{\text{вх}}$. Мікроконтролер через порт P3, що працює в режимі виведення, передає двійкові коди в ЦАП, вихід якого під'єднується до входу неінвертувального компаратора. Вихід компаратора (двійковий сигнал) під'єднується до входу процесора P1.0.

Програма аналого-цифрового перетворення «працює» таким чином: МК видає через порт P3 байт даних, який перетворюється ЦАП на аналоговий сигнал $U_{ЦАП}$, що порівнюється зі вхідним аналоговим сигналом $U_{вх}$ у компараторі, вихідний сигнал якого (результат порівняння) потім аналізує процесор. Залежно від значення сигналу на вході P1.0 МК або залишає старший біт байта, що виводиться в одиницю, якщо $U_{ЦАП} < U_{вх}$, або скидає його в нуль, якщо $U_{ЦАП} > U_{вх}$. Потім аналогічним чином у порядку зменшення вагових значень перевіряється кожен біт байта, що виводиться. Цей фрагмент алгоритму повторюється n разів, де n – кількість двійкових розрядів інтегрального ЦАП.

Перетворення є диференціальним, точність становить $\pm 0,5$ одиниці LSB.

Структурну схему методу послідовних наближень зображено на рис. 3.12. Напруга на виході DAC утворюється за формулою

$$U_{DAC} = \frac{U_{REF}}{2^N} (E^7 \cdot 2^7 + E^6 \cdot 2^6 + \dots + E^0 \cdot 2^0);$$

$$K = E^7 \cdot 2^7 + E^6 \cdot 2^6 + \dots + E^0 \cdot 2^0,$$

де N – кількість розрядів коду; K – двійковий код числа; E – двійкові біти коду.

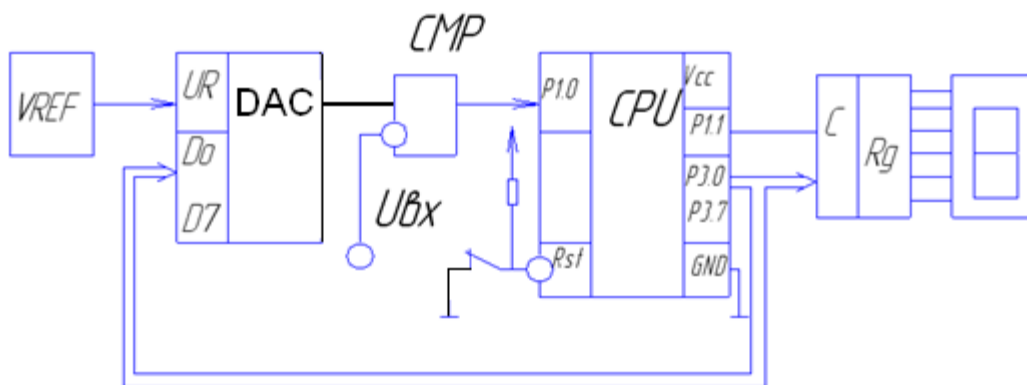


Рис. 3.12. Структурна схема методу послідовних наближень

Алгоритм:

1. Установити поточний (починаючи зі старшого) біт коду K таким, що дорівнює одиниці $\Rightarrow U_{DAC} = U_{REF} / 2^n \times K$, $U_{REF} \geq U_{вх} \geq 0$. Якщо біт P1.0 = 1, то записати до старшого розряду 0, інакше – 1.

2. Якщо всі розряди коду проаналізовано, то записати отримане значення K у зовнішній регістр пам'яті й вийти з програми, інакше – перейти до наступного вагового розряду коду й повернутися до п. 1.

Програма реалізації алгоритму послідовного наближення на мікроЕОМ родини INTEL 8051 може мати такий вигляд:

```

CMP_Q    BIT  P1.0      ; бит - результат порівняння (вихід
компаратора)
START:    MOV  R0, #0    ; скидання нагромаджувача даних
          MOV  R1, #80h  ; MASKA
NEXT:     MOV  A, R0     ;
          ORL  A, R1     ;
          MOV  P3, R1    ; DAC
          JNB  CMP_Q, WEITER
          XRL  A, R1     ; інвертування, якщо  $U_{вх} < E_{оп}$ 
WEITER:   MOV  R0, A     ; повернути нагромаджувач
          MOV  A, R1     ; змінити маску
          RRC  A         ;
          MOV  R1, A     ;
          JNZ  NEXT      ; якщо маска є меншою за 0, то перейти до
наступного біта
          MOV  P3, R0    ;

```

Метод подвійного інтегрування

Схему під'єднання до МК додаткової апаратури показано на рис. 3.13. Перетворення виконують відповідно до формули

$$\int_0^{t_0} U_x(t) dt = \int_0^{t_x} -E_{оп}(t) dt.$$

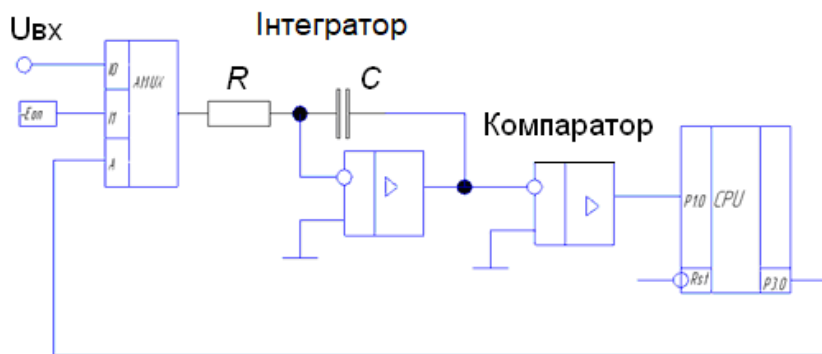


Рис. 3.13. Структурна схема АЦП подвійного інтегрування

Спочатку до входу інтегратора подається негативна опорна напруга $-E_{оп}$, яка надходить туди зі входу аналогового мультиплексора AMUX D1 (за наявності потенціалу логічної одиниці на адресному вході мультиплексора A). При цьому на виході інвертувального інтегратора через деякий час буде встановлено високий рівень, а на виході компаратора сформовано

нульовий логічний сигнал (вихід інтегратора під'єднано до інвертувального входу компаратора).

Процес перетворення складається з двох етапів. Спочатку відбувається інтегрування вимірюваного вхідного сигналу $U_{\text{вх}}$ протягом певного часу T_1 . Відлік інтервалу T_1 починається з моменту t_0 переходу напруги на виході інтегратора через нуль. (Цей факт визначається станом входу P1.0 мікропроцесора, з'єднаного з виходом компаратора). У цій схемі напруга вхідного сигналу має бути позитивної полярності. Потім на момент часу t_1 на вхід інтегратора подається опорна напруга $-E_{\text{оп}}$ (протилежної полярності), що виконується програмним змінням логічного потенціалу на адресному вході AMUX, і вимірюється час інтегрування T_2 . (У момент закінчення циклу інтегрування логічний потенціал на виході компаратора знову стане нульовим.) За умови рівності зарядів $|E_{\text{оп}} T_2| = U_{\text{вх}} T_1$ знаходиться вимірювана величина $U_{\text{вх}} = E_{\text{оп}} T_2 / T_1$.

Час T_1 (період першого інтегрування) вибирають так, щоб при максимальній вхідній напрузі ($U_{\text{вх.макс}} = |-E_{\text{оп}}|$) інтегратор не ввійшов у стан насичення. Програма підтримки АЦП може мати такий вигляд:

; Перетворювач подвійного інтегрування *** версія для Intel-8051

Q_CMP	BIT	P1.0	; вхід процесора
A_MUX	BIT	P3.0	; вихід процесора
T1	EQU	3476h	; час T_1 (у циклах CPU)
	MOV	TMOD,#01H	; настрій T/C ₀ на режим 16 бітів
	MOV	TH0,#HIGH(NOT(T1)+1)	; завантаження T/C ₀
	MOV	TL0, #LOW(NOT(T1)+1)	;
	SETB	Q_CMP	; настрій P1.0 на введення даних
	SETB	A_MUX	; подання $-E_{\text{оп}}$ на вхід інтегратора
WAIT:	JB	Q_CMP, WAIT	; очікування появи негативного рівня CMP
	CLR	A_MUX	; подання $U_{\text{вх}}$ на вхід інтегратора
WAITT0:	JNB	Q_CMP, WAITT0	; очікування моменту часу початку T_0
	SETB	TCON.4	; старт лічильника часу T_1 (T/C ₀)
WAITT1:	JNB	TCON.5, WAITT1	; очікування моменту T_1
	SETB	A_MUX	; подання $-E_{\text{оп}}$ на вхід інтегратора
WAITT2:	JB	Q_CMP, WAITT2	; очікування моменту T_2
	CLR	TCON.4	; стоп T/C ₀ (CLR TR ₀)
	CLR	TCON.5	; скидання прапорця TF ₀ (CLR T F ₀)
	MOV	B, TH0	; занесення результату до
	MOV	A, TL0	; регистрової пари BA

Програма дає змогу сформувати 16-бітний код, еквівалентний вхідній напрузі, що свідчить про високу роздільну здатність методу перетворення.

Недолік методу – змінення часу робочого циклу залежно від рівня вхідної напруги. Переваги: висока роздільна здатність і завадостійкість.

Інтегральні перетворювачі за методом напруга/частота (ПНЧ – VFC)

У методі АЦП передбачено наявність деяких апаратних засобів, що мають назву перетворювача напруги/частоти (ПНЧ) (наприклад, AD654, AD7741, LM331, VFC32, K1108ПП1 і т. ін.), які на своєму виході ставлять у відповідність до вхідної напруги безперервну імпульсну послідовність, частота якої є прямо пропорційною до вхідної напруги з деяким коефіцієнтом, що має назву коефіцієнта перетворення $[K] = V / Hz$.

Такі пристрої працюють у частотному діапазоні від 0 до 500 KHz при вхідних напругах від 0 до $+U_{ж}$. Нелінійність перетворювачів зазвичай є не більшою за 0,2 %, температурна й довготривала стабільність визначаються якістю зовнішніх елементів, до яких належать конденсатор C і струмообмежувальний резистор R .

Алгоритм ПНЧ:

1) перетворити напругу на частоту проходження імпульсів з деяким коефіцієнтом K : $F_x = K \cdot U_x$;

2) виміряти частоту з допомогою вимірювача частоти, реалізованого на базі мікроЕОМ.

Схему алгоритму перетворення напруга/частота показано на рис. 3.14.

Склад частотоміра: ФОВ – формувач опорного часу (наприклад, T_1), CR – шістнадцятирозрядний лічильник кількості імпульсів протягом опорного інтервалу часу, M_x – вихідний мультиплексор даних.

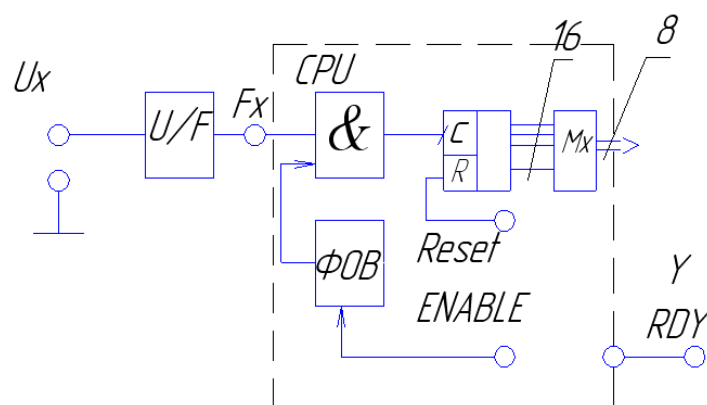


Рис. 3.14. Структурна схема алгоритму перетворення типу напруга/частота

Програмна підтримка перетворення за методом ПНЧ:

TIME EQU	NOT (50000)	;	для F = 6 MHz; t = 2μS
MOV	TMOD, #51h	;	обидва пристрої перевести в режим CT16
MOV	TH0, HIGH (#TIME);		T ₀ – таймер опорного часу
MOV	TL0, LOW (#TIME) ;		T ₁ – лічильник імпульсів
MOV	TL1,#00	;	очищення лічильника імпульсів
MOV	TH1,#00	;	
CLR	TF0	;	скидання прапорця переповнення TIMER_0
SETB	EA	;	спільний дозвіл переривань
MOV	TCON,#50	;	пуск обох лічильників одночасно
M1: JNB	TF0, M1	;	очікування опорного часу
CLR	TR1	;	зупинити лічильник імпульсів
CLR	TR0	;	зупинити таймер опорного часу
MOV	P1,TL0	;	передати назовні молодший байт напруги
MOV	P2,TH0	;	передати назовні старший байт напруги

Порядок виконання роботи

1. Вивчити алгоритми й програмні методи підтримки пристроїв перетворення даних.
2. Записати програми перетворення мовою C й відлагодити їх у середовищі WinAvr або ARDUINO IDE.
3. Після виконання ПР скласти звіт, де навести розроблені програми й відповіді на контрольні запитання.

Контрольні запитання

1. Наведіть основні характеристики й параметри АЦП.
2. Що розуміють під точністю, а що – під роздільною здатністю АЦП?
3. Чим різняться основні методи АЦ перетворення?
4. Яке значення у процесі перетворення має мікроконтролер?
5. Від чого залежать параметри INL і DNL аналого-цифрового перетворювача?
6. Як оцінюють шумові показники АЦП?
7. Який тип перетворювачів називають інтегрувальними?
8. Що розуміють під гармонічними спотвореннями цифроаналогового перетворювача?
9. Які методи підвищення точності АЦП ви можете запропонувати?
10. Як здійснюється передання цифрової інформації від АЦП до МП?
11. Запропонуйте методи поліпшення завадостійкості АЦП.

Практична робота № 4

РАСТРОВІ ФАЗОВІ ПЕРЕТВОРЮВАЧІ ПЕРЕМІЩЕНЬ ФПП З ПІДРАХУНКОМ МУАРОВИХ СМУГ (НАКОПИЧУВАЛЬНОГО ТИПУ)

Мета роботи: вивчити структури й принципи дії оптичного інкрементального квадратурного перетворювача переміщень і методи розроблення програмного забезпечення до нього.

Завдання: розробити алгоритм і програму відліку кількості силового розмірного переміщення інструмента верстата з ЧПК.

Загальні відомості

Основним вузлом будь-якого растрового фазового перетворювача переміщень (ФПП) є оптичний модулятор, що складається з вимірювального (радіального або лінійного) та індикаторного (нерухомого) елементів. За типом застосовуваного сполучення розрізняють растрові, дифракційні й інтерференційні ланки [1].

Між переліченими ланками є багато спільного, оскільки в усіх є спільні ознаки і періодична структура. Переміщення вимірювального елемента відносно індикаторного на деяку величину спричиняє повторення показань вихідного сигналу чутливого елемента (фотоприймача). Найменша величина переміщення, коли його показання повторюються, є періодом ланки.

Цілком аналогічним поняттю характеристики пропускання растрового сполучення є поняття характеристики розподілу освітленості в інтерференційному полі інтерференційної ланки.

При поєднанні штрихових растрів утворюються комбінаційні фігури (смуги), крок, форма і напрям яких визначаються взаємним розташуванням штрихів сполучених растрів.

При поєднанні двох систем хвиль утворюються інтерференційні фігури (інтерференційні смуги), крок, форма і напрям яких визначаються взаємним розташуванням хвильових структур.

Залежно від характеру променистого потоку розрізняють пропускні й відбівні растри. Пропускні растри є системою прозорих і непрозорих елементів, відбівні виконуються у вигляді ґрат з елементами, що дзеркально відбивають світло [1]. Принцип дії растрових ФПП полягає в такому.

Під час кутового або лінійного переміщення вимірювального растра відносно нерухомого індикаторного растра відбувається модуляція світлового потоку за амплітудою у функції переміщення, тобто періодичне змінення освітленості чутливого елемента, розташованого за полем растрового сполучення.

При цьому освітленість фотоприймача є максимальною при збігу штрихів вимірювального й індикаторного растрів і мінімальною, коли штрихи одного растра закривають прозорі ділянки іншого. Повний цикл змінення освітленості фотоприймача відбувається під час переміщення растра на величину кроку. Підрахуванням кількості таких циклів можна вимірювати лінійні й кутові переміщення з точністю до кроку растра.

Якщо розташувати растри так, щоб між штрихами утворився деякий кут, то виникнуть чітко видимі широкі темні й світлі смуги, які мають назву комбінаційних (або муарових). Переміщення вимірювального растра відносно індикаторного в напрямку, перпендикулярному до штрихів, на величину кроку растра викличе переміщення комбінаційних смуг у поперечному напрямку (вздовж штрихів) на величину кроку смуг. При цьому крок муарової лінії пов'язаний з кроком растра таким співвідношенням [1]:

$$G = q/2\sin(\varphi/2),$$

де G – крок муарової смуги; q – крок растра; φ – кут зсуву вимірювального растра відносно індикаторного.

Аналогічно (як у разі паралельного розташування растрів) підрахунком кількості муарових смуг можна вимірювати переміщення з точністю до кроку растра.

Описаний принцип дії дає змогу будувати досить прості схеми растрових ФПП, наприклад, із підрахунком кількості муарових смуг. Одну із схем такого ФПП нагромаджувального типу зображено на рис. 3.15. Розглянемо принцип її дії.

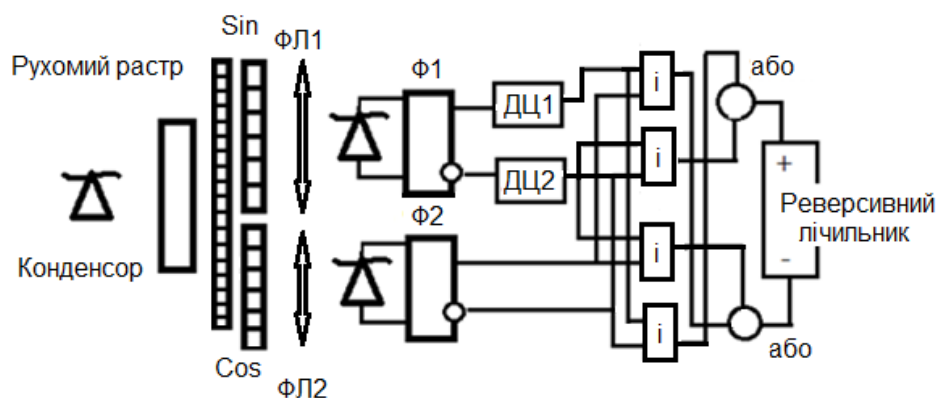


Рис. 3.15. Структура растрового квадратурного перетворювача переміщень нагромаджувального типу

Для того щоб забезпечити реверсивне лічення, растровий модулятор (РМ), що складається з конденсора 1, растрових решіток РР(SIN, COS), діафрагми Д і двох об'єktivів 2 (ФЛ1, ФЛ2), має видавати два сигнали, зсунутих за просторовою фазою на $\pi/2$ один відносно одного, тобто таких, що знаходяться у квадратурі.

Це досягається відповідним просторовим (кутовим або лінійним) зсувом індикаторних растрів РР один відносно одного на відстань $n\lambda + \lambda/4$, де λ – відстань між штрихами растра. Від фотоприймачів ФПП1 і ФПП2 синусоїдальні сигнали I і II, зсунуті за просторовою фазою на $\pi/2$, надходять на формувачі Ф1 і Ф2. Кожен із формувачів має по два виходи. З одного виходу формувача знімається прямокутна напруга з тією ж фазою, що й синусоїдальний вхідний сигнал, а з іншого виходу знімається інвертований сигнал (протифаза). Унаслідок цього маємо чотири прямокутні напруги А, В і С, D, три з яких є зсунутими відносно кожного попереднього на $\pi/2$ за просторовою фазою. Вихідні сигнали А і В подаються на диференціювальні ланцюги ДЦ1 і ДЦ2. Продиференційовані імпульси А' і В' надходять на відповідні входи схем збігу І1 – І4, на інші входи яких подаються відповідні сигнали з виходів Ф2. Для показаної на схемі комутації входів і виходів схем збігів І1 – І4 імпульси виробляються на шині прямого ходу й подаються на вхід (+) реверсивного лічильника РЛ, якщо вимірювальний растр рухається в прямому напрямку. Під час руху в протилежному напрямку лічильні імпульси виникають на шині зворотного ходу і потім подаються на вхід (–) РЛ. Під час переміщення растра на один крок на лічильник РЛ подаються два імпульси, тобто розглянута схема ФПП має роздільну здатність, що відповідає $1/2$ кроку растра. Принцип дії схеми ілюструється функціональною схемою й часовими діаграмами роботи (рис. 3.16).

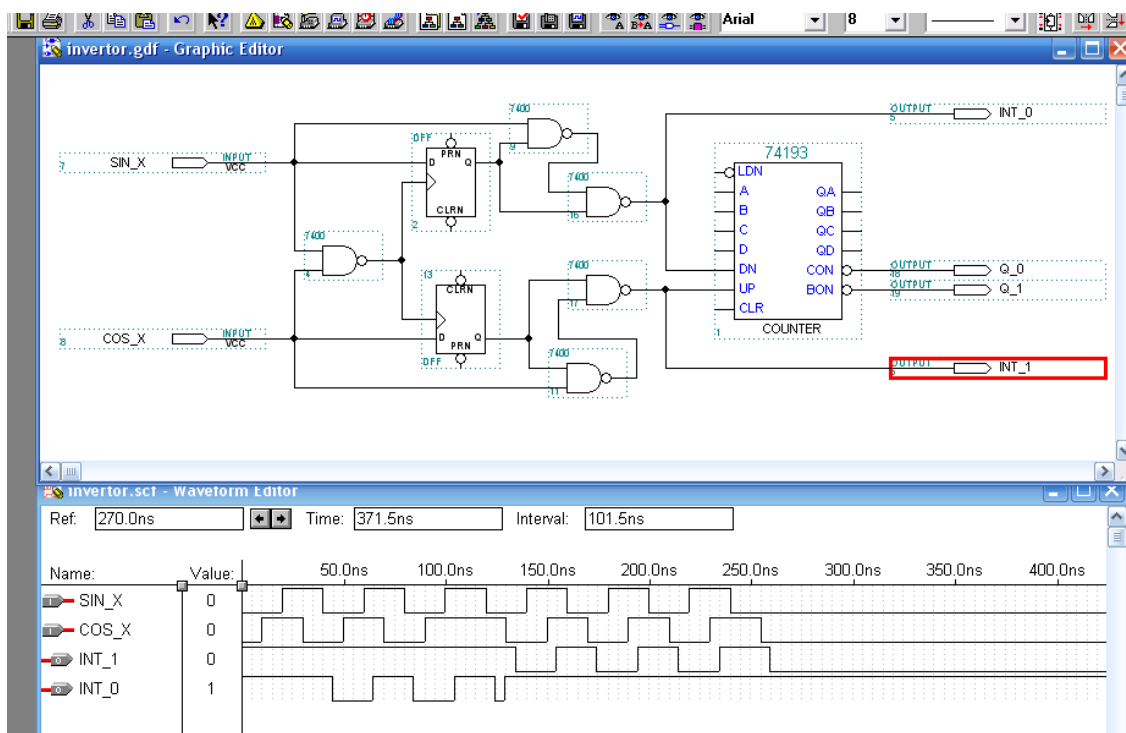


Рис. 3.16. Структурна схема фазового детектора-розрізнявача напрямку руху: функціональна логічна схема (зверху); часові діаграми роботи (знизу): EPLD EPM3032ALC44 (ALTERA)

Ускладненням блока отримання лічильних імпульсів можна збільшити роздільну здатність до $1/4$ і навіть до $1/8$ кроку растра (тобто в 4 рази). Ці схеми є відомими під назвою інтерполяційних і тут не наводяться. Зазначимо тільки, що лінійні й цифрові інтегральні мікросхеми (ІМС), що випускаються сьогодні промисловістю, дають змогу створювати растрові нагромаджувальні ФПП з $1/8$ кроку, які задовольняють усі вимоги щодо точності інтерполяції й надійності роботи. При цьому отримують роздільну здатність в 1–2 мкм для растрів з кроком 8 мкм і діленням кроку растра на 4 або 8 частин (рис. 3.17).

Розпіновка

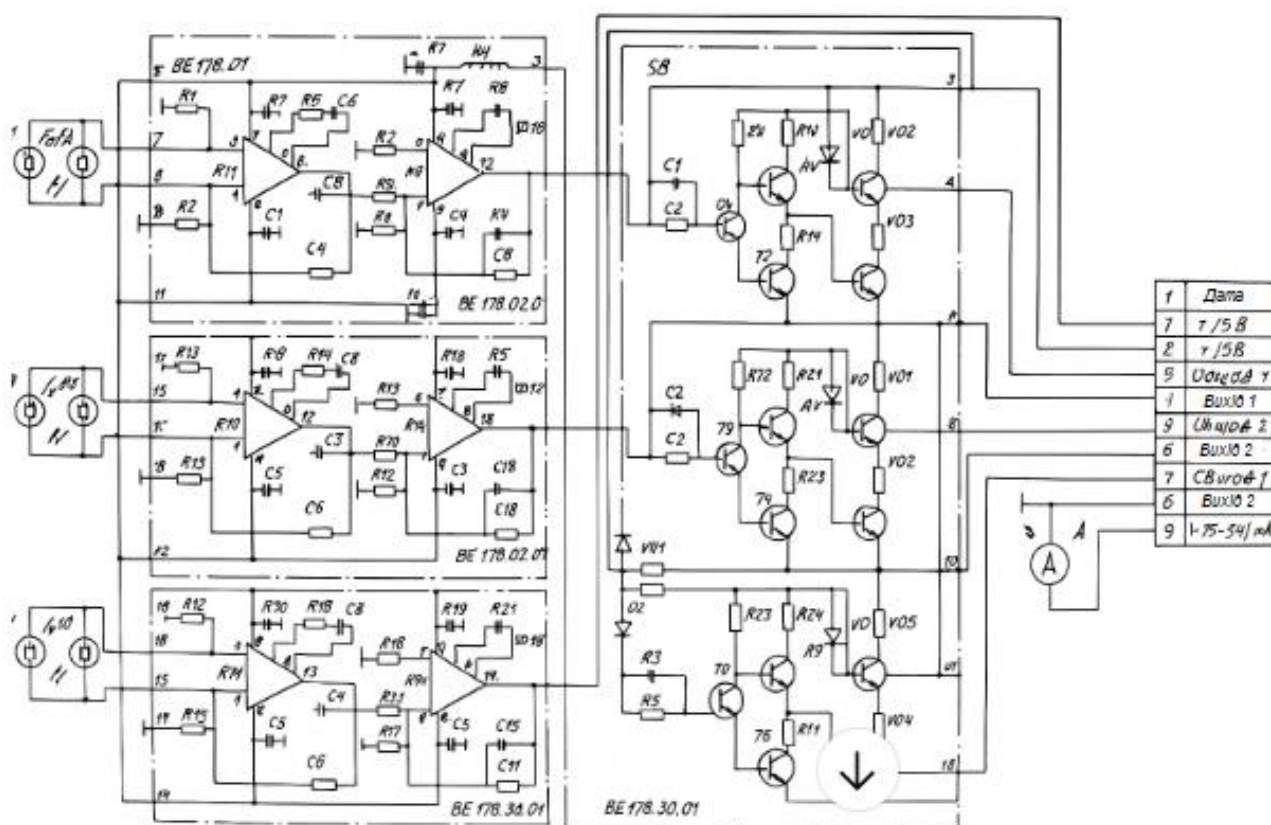


Рис. 3.17. Функціональна електрична схема ФПП кутових переміщень BE178-A5

Одним зі способів побудови растрових нагромаджувальних ФПП з роздільною здатністю, що дорівнює десятим часткам мікрметра, є застосування проміжної інтерполяції на електронно-променевій трубці.

Порядок виконання роботи

1. Вивчити принципи побудови растрових перетворювачів переміщення на основі ФПП BE178-A5.
2. Розробити схему фоторастрового інкрементального квадратурного енкодера для цифрового керування настроєнням кроку вимірювального

перетворювача лінійних (кутових) розмірних переміщень (генератора стандартних сигналів – ГСС) з уповільненням швидкості нагромадження в 10, 25, 50, 100 разів з індикацією кроку й розміру (частоти вихідного сигналу).

3. Змодельювати роботу енкодера в середовищі Max + Plus II або MATLAB.

4. Розробити програму підтримки нагромадження даних на базі МП платформи *ARDUINO* в середовищі *ARDUINO IDE* за зразком, наведеним у дод. 1.

5. За підсумками роботи оформити звіт, де розмістити результати розроблення й моделювання технічної системи.

Контрольні запитання

1. Що розуміють під оптичним інкрементальним квадратурним енкодером?

2. Перелічіть можливі сфери застосування інкрементальних енкодерів.

3. Який зміст мають терміни «квадратурний» і «реверсивний»? Де можна застосувати нереверсивні енкодери?

4. Яку функцію в системі оброблення виконує фазовий детектор?

5. Яка функція в системі оброблення відводиться контролеру?

6. Чому в Ардуїно-скетчі під час оброблення даних використовується режим переривань, а не програмний режим обміну?

7. Чим визначається точність роботи оптичного квадратурного енкодера?

8. Поясніть принцип роботи МП-системи в режимі переривань і його призначення.

9. Чим відрізняються оптичні квадратурні енкодери від резистивних, наприклад Sparkfun, або EC11?

4. ПРОЄКТУВАННЯ МІКРОПРОЦЕСОРНИХ СИСТЕМ АВТОМАТИКИ

Практична робота № 5

ПРОГРАМУВАННЯ ПОРТІВ УВЕДЕННЯ/ВИВЕДЕННЯ МІКРОПРОЦЕСОРА

Мета роботи: вивчення методів програмування портів введення/виведення МП.

Завдання: запрограмувати біти PB0 та PB2 порту *B* процесора *AtMega32* на виведення інформації та ввести її через виводи PB4 і PB6 того ж МП-пристрою. Перевірити правильність передання даних.

Теоретичні відомості

Порт мікропроцесора AVR складається з трьох 8-розрядних регістрів:

- $DDRx$ – регістр напрямку передання даних (1 – виведення, 0 – увведення);

- $PORTx$ – регістр даних, що виводяться з МП назовні (передання);

- $PINx$ – регістр даних, які надходять іззовні до МП.

Літера «x» символізує один з чотирьох портів процесора, які позначаються літерами A, B, C, D.

Якщо в порт $DDRC$ записати всі одиниці, тобто $DDRC = 0b11111111$, то всі виводи порту C буде призначено для виведення інформації з процесора на зовнішні пристрої, а якщо в порт $DDRD$ записати число $0xA6 = 0b10100110$ ($DDRD = 0xA6$), то виводи $PD1$, $PD2$, $PD5$, $PD7$ буде призначено для виведення даних, а інші ($PD0$, $PD3$, $PD4$, $PD6$) – для зчитування (увведення) даних у МП.

Порядок виконання роботи

1. Запишіть програму (лістинг 1) у середовищі *AVR Studio* та скопіюйте її. Перевірте її роботоздатність шляхом відлагодження у середовищі *AVR Studio*.

2. Зберіть схему МП пристрою за рис. 4.1. (Якщо немає *AtMega32*, то скористуйтеся платформою *ARDUINO Mega 2560*, або *ARDUINO UNO*.)

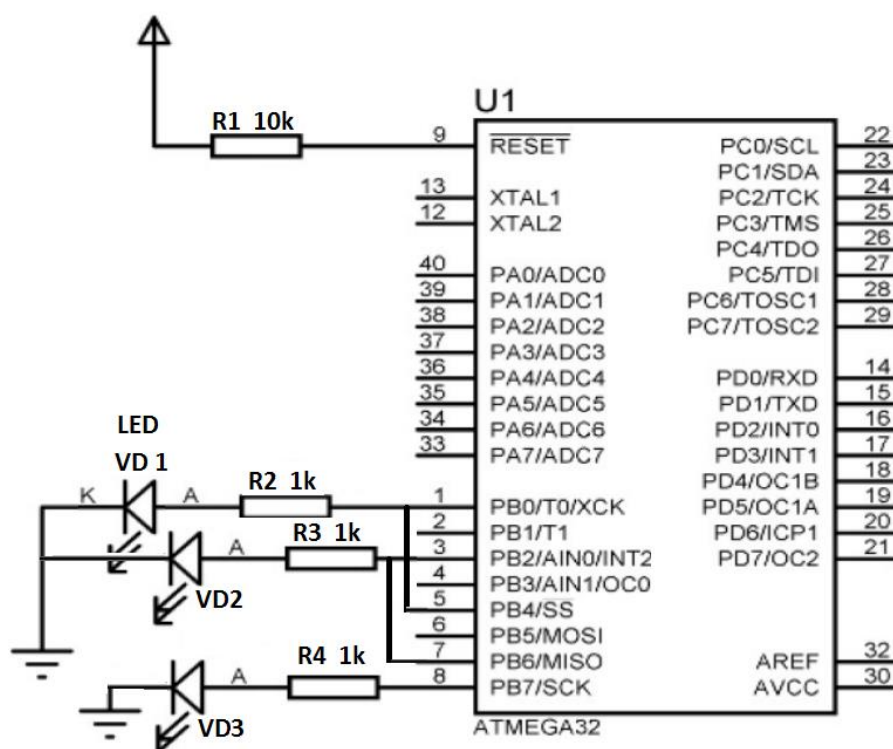


Рис. 4.1. Схема електрична функціональна МП-світлодіодного дисплея

3. Проведіть програмування МП і перевірте роботу схеми.

4. Розробіть програму виведення в порт *B* власного порядкового номера за списком групи й перевірте правильність його введення зчитуванням даних через порт *C*. У разі правильного виконання завдання виведіть сигнал логічної одиниці в порт *PA0*.

5. Оформіть звіт про виконання практичної роботи.

Лістинг 1. Програма–приклад керування портами 8-розрядного мікропроцесора сім'ї AVR: у разі правильного виведення й зчитування даних загориться світлодіод *VD3* :

```
/*-----  
This program demonstrates the writing and reading from port  
Author : WRITE YOUR NAME and Date hear: USER, 04.12.2024, gr.249  
LED connected to PB1, PB3 and PB7. PB7 -check pointer.  
PB7 -1 PB6-0 PB5-0 PB4-0 PB3 -1 PB2-0 PB1-1 PB0-0  
-----*/  
#include <avr/io.h> // standard AVR header  
int main(void)  
{  
    int i=100; // time delay  
    DDRB = 0b10001111; // setting PB0...PB3, PB7 as output, PB4...PB6 as input  
    PORTB = 0b00001010; // setting value for PORTB  
next: // label  
    if (i>0) {i--; goto next;}; // delay timer  
    if ((PINB4) && (PINB6)) PORTB|= 1<<PB7; // if pin4=1 and pin6=1) => PB7=1  
    else PORTB=0;  
    while (1); // indefinite loop  
}
```

Практична робота № 6

ПРОГРАМУВАННЯ ЧАСОВИХ ІНТЕРВАЛІВ ІМПУЛЬСНИХ СИГНАЛІВ

Мета роботи: вивчення методів відтворення імпульсних сигналів певної частоти й пористості.

Завдання: запрограмувати лінію PD1 процесора *AtMega32* на виведення інформації та здійснити через цей інформаційний канал передання перших трьох знаків (літер) свого прізвища в коді Морзе зі швидкістю 60 знаків за хвилину.

Теоретичні відомості

Формування точних часових інтервалів пов'язане з інтервальними таймерами-лічильниками МП і частотою тактової синхронізації процесора F_{cpu} , яка задається директивою `define F_CPU` у Hz. Директива `include`

<util/delay.h> підключає бібліотечну функцію формування часової затримки в мікросекундах. Функція `_delay_loop2`, яку підключає заголовок <util/delay.h>, може формувати часовий інтервал від 1 до 262000 мікросекунд, який задається параметром цієї функції у вигляді беззнакового 16-розрядного цілого числа (`uint16_t`). Для формування довших інтервалів введено програму `PAUSE()`, яка формує затримку до 2.55 S з роздільною здатністю 10 мс. У прикладі необхідно звернути увагу на використання операторів логічного «або» (`()`) і логічного «і» (`&`), а також на використання оператора «ні» (`!`) у підпрограмах формування крапки (DOT) і тире (DASH). Відповідність часу затримки функції `_delay_loop2()` реальному відліку часу відбувається тоді, коли частота кварцевого резонатора, встановленого в МП-системі, точно відповідає тому, що визначено директивою `#define F_CPU XXX`.

Порядок виконання роботи

1. Зібрати схему за рис. 4.2, де вихід PB0 МП під'єднати до бузера зі встроєним генератором. Записати програму прикладу в середовищі *AVR Studio* та перевірити її роботоздатність.

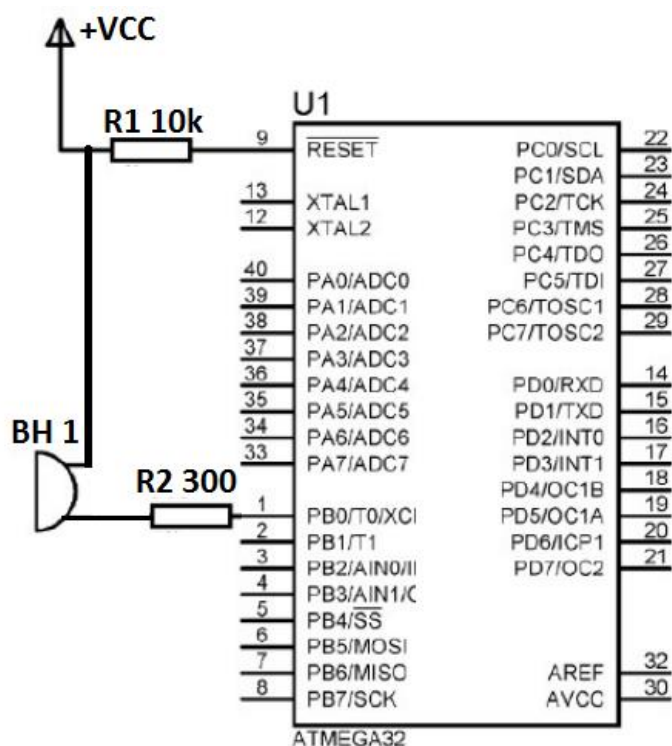


Рис. 4.2. Схема електрична функціональна МП-системи відтворення звукових сигналів

2. Розробити програму виведення в порт *B* власного прізвища (перші три літери) азбукою Морзе (кодування букв латиницею Ж = V, Щ = Q, Й = J).

3. Виконати програмування процесора *AtMega32* (або платформи *ARDUINO*) і перевірити коректність роботи пристрою під'єднанням бузера (BH1) з комплекту оснащення лабораторної платформи.

4. Розрахувати середній час передання символу, який має становити 1 с (увесь час передання – 3 с). Між знаками передбачити паузи.

5. Оформити звіт про виконання роботи.

Лістинг 2. Програма-приклад формування часових інтервалів і використання логічних операцій «and», «or», «not» та виклику підпрограм:

```
#include <avr/io.h>          // standard AVR header
#include <util/delay.h>      // AVR delay header
#define F_CPU 1000000       // Quartz Oscillator definition (Hz)

void PAUSE(uint8_t t)        // Delay subroutine
{
    if (t>0)
    { next:
      _delay_loop_2(10000);
      t--;
      if (t>0) goto next;
    }
}

void DOT(void)               // Dot subroutine
{
    PORTB |= 1<<PB0;         // PB0 = 1
    PAUSE(5);                // Delay 50 mS
    PORTB &= !(1<<PB0);      // PB0 = 0
    PAUSE(3);                // Delay 30 mS
}

void DASH(void)              // Dash subroutine
{
    PORTB |= 1<<PB0;         // Setting PB0
    PAUSE(7);                // Delay 70 mS
    PORTB &= !(1<<PB0);      // Clear PB0
    PAUSE(3);
}

int main(void)
{
    DDRB|=(1<<PB0);          // PB0 - as output
    while (1)                // indefinite loop
    {
        DOT();
        DOT();               // «S»
        DOT();
        PAUSE(7);            // Delay 70 mS
        DASH();
        DASH();              // «O»
        DASH();
        PAUSE(7);            // Delay 70 mS
    }
}
```

```

DOT ( ) ;
DOT ( ) ;           //  «S»
DOT ( ) ;
PAUSE (30) ;         // Delay 300 mS
}
}

```

Контрольні запитання

1. Що розуміють під таймером-лічильником МП системи?
2. Які функції та в яких режимах виконує МП-системний таймер?
3. Чим відрізняється програмний та апаратний системні таймери?
4. Яким числом обмежено часовий інтервал роботи таймера і чим він визначається?
5. У який спосіб програмується затримка в часі і чим визначається її точність?
6. Чи можна підвищити точність роботи системного таймера до вторинного еталона частоти ($\Delta f/f = 10^{-11}$)?

Практична робота № 7

ПРОГРАМУВАННЯ МП КЕРУВАЛЬНОГО АВТОМАТА

Мета роботи: вивчення методів програмування системи автоматичного керування.

Завдання: за прикладом автомата керування форвакуумним насосом (Лістинг 3, рис. 4.4, схему алгоритму керування яким зображено на рис. 4.3) розробити програму пуску/зупинення електроприводу головного руху верстата ЧПК за такими вимогами:

1) при надходженні від ЧПК команди *M1* (ліве обертання) процесор очікує 2 с, поки двигун набере номінальних обертів. Якщо після закінчення затримки обертовий сенсор не спрацьовує, то напруга на електропривод головного руху знімається, а на ЧПК видається відповідь «Відмова верстата за *M*» у протилежному випадку на ЧПК видається сигнал «Підтвердження *M*». Такі ж дії відбуваються за командою *M2* (праве обертання). При знятті команди *M1* або *M2* процесор запускає програму гальмування противмиканням двигуна головного руху на 2 с. При цьому відбувається швидке гальмування двигуна, після чого напруга з нього знімається повністю та знімається сигнал «Підтвердження *M*» до ЧПК. Двигун вмикається за умови наявності трифазної первинної напруги змінного струму й струму збудження у відповідному контурі двигуна. Якщо напруги або струму збудження немає, то формується сигнал «Відмова верстата за *M*»;

2) розробити МП-систему керування автоматом заміни інструменту токарного верстата з ЧПК: при наданні сигналу Т1–Т6 на токарний верстат з боку системи ЧПК, верстат виконує чотири послідовні операції :

- а) знімає сигнал «Відповідь за Т» до ЧПУ;
- б) знімає інструментальну револьверну головку (РГ) з фіксувального упору (за датчиком фіксації);
- в) прокручує барабан РГ проти ходу годинникової стрілки до заданої в номері інструменту позиції;
- г) ставить РГ на фіксувальний упор (за датчиком фіксації);
- д) установлює сигнал «Відповідь за Т» до ЧПК, який не виконує наступного кадру КП до його надходження.

Теоретичні відомості

Структурне проєктування програмного забезпечення

Цей метод створення програмного забезпечення (ПЗ) полягає у розподіленні об'єкта автоматизації на окремі блоки (агрегати), кожен з яких є окремим фізичним об'єктом технічної системи (ТС), що має певні ознаки, технічні властивості й виконує певні технологічні функції. До цього об'єкта, виходячи з принципів його використання, наведених у технічному завданні (ТЗ), розробляються лінгвістичні правила, які формалізують його взаємодію з іншими складовими ТС. Формалізований опис алгоритму роботи об'єкта є проміжним кроком між ТЗ і програмним забезпеченням, яке відтворює його з використанням певної мови програмування МП пристрою.

Розглянемо приклад розроблення ПЗ до модуля керування форвакуумним насосом (ФВН, *FVP*) дугової системи нанесення іонно-плазмових покриттів у вакуумі. Система складається з багатьох технічних агрегатів, кожен з яких є її структурним об'єктом і потребує модульної алгоритмічної та програмної підтримки, взаємодія між якими (згідно з вимогами ТЗ) відбувається у межах модульного програмного забезпечення. ТЗ роботи ФВН надається описовим алгоритмом, у якому передбачено можливості його роботи в трьох режимах експлуатації: налагодження (*Manual*), автоматичному (*Automat*), аварійному (*Alarm*):

1) ФВН можна увімкнути лише за умови зачиненої двері вакуумної камери (кінцевий вимикач $SQ2 = 1$);

2) для перемикання системи з режиму *Manual* (1) в режим *Automat* (0) задіяно перемикач $SA1$;

3) в автоматичному режимі, коли немає тиску первинного вакууму (сенсор $SP1 = 0$) у форвакуумному ланцюзі протягом 10 с, починаючи від моменту вмикання, ФВН необхідно вимкнути через пошкодження форвакуумної магістралі. Має бути сформовано сигнал «*FVP_Alarm*»;

4) за умови встановлення перемикачки $SQ1$ «*Test*», пункт 3 не виконується, а ФВН вмикається шляхом натиснення кнопки $SB1$ «*Start FVP*» (з фіксацією). Перемикач $SQ1$ у цьому режимі встановлено;

5) у режимі автоматичного керування передбачається, що ФВН остаточно зупиняється при вимиканні не лише кнопки SB1, але й датчика температури масла: $T_M < 55^\circ\text{C}$ ($SQT = 0$);

6) при надходженні зовнішнього сигналу «Alarm» насос не може бути ввімкнутим у жодному з передбачених варіантів роботи;

7) за умов штатної роботи насоса сигнал «FVN_Alarm» має бути вимкненим.

На рис. 4.3 зображено граф бінарного алгоритму керування форвакуумним насосом у всіх передбачених режимах його використання згідно з описовим алгоритмом. Символом Dz визначено сигнал пуску таймера затримки 10 с, а Tz – сигнал переповнення таймера. На базі технічного завдання систему керування можна задати системою логічних рівнянь:

$$FVP = !Alarm\{Sq1*Sb1 + !Sq1*!Sa1*Sq2(Sqt+Sb1)[Tz*Sp1 + !Tz*!Sp1]\};$$

$$DZ = !Alarm*!Sq1*Sq2*!Sa1*Sb1*!Tz.$$

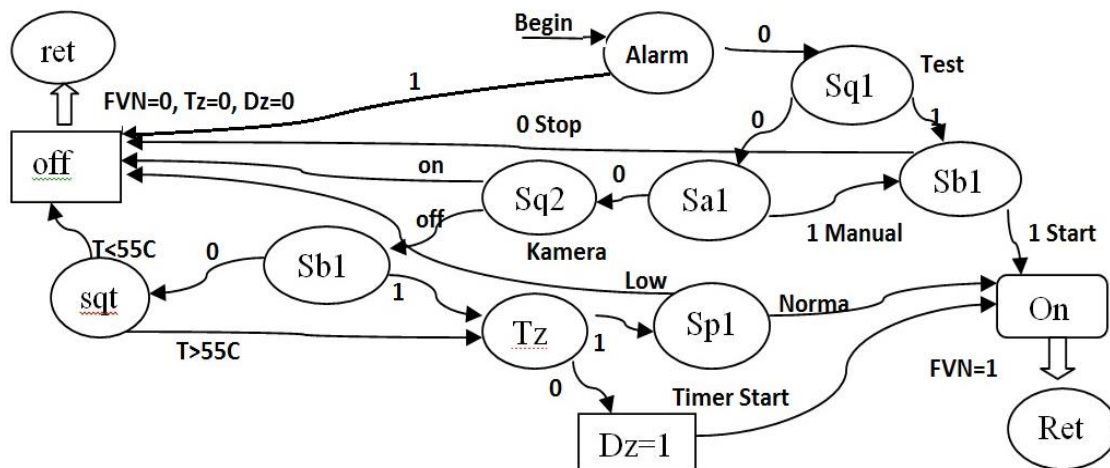


Рис. 4.3. Граф бінарного алгоритму керування форвакуумним насосом

Найпростішим методом написання програми керування є метод заміни кожної вершини графа бінарного алгоритму відповідним програмним оператором будь-якої мови програмування. Якнайкращою буде мова, оснащена булевим процесором, наприклад мова асемблера *Intel 8051*. Оператори *JB flag*, *Addr*, *JNB flag*, *Addr* цієї мови об'єднують дію трьох операторів: зчитування булевої змінної (*flag*), її перевірка й перехід за адресою *Addr* при виконанні умови, зазначеної в операторі, *JB* (або *flag = 1*), *JNB* (або *flag = 0*). За умови використання операторів булевого процесора програмну реалізацію алгоритму керування ФВН можна записати мовою асемблера *Intel 8051* [5] у такому вигляді:

```
FVP: JB      Alarm, OFF_FVP ; Перехід за аварією
      JB      SQ1, L1       ; Перехід за наявності перемички «Test»
```

```

JB    SA1, L1          ; Перехід на ручне керування
JNB   SQ2, OFF_FVP     ; Перехід при відчиненій камері
JB    SB1, L2          ; Перехід за вмиканням в автоматичному режимі
JB    SQT, L2          ; Невимкнення за умови високої температури
OFF_FVP:
CLR   FVP              ; Відімкнення насоса
CLR   TF0              ; Повернення таймера до вихідного стану
CLR   TR0              ; Зняття сигналу збудження таймера
MOV   TL0, #LOW(TIME)  ; Перезавантаження часу затримки
MOV   TH0, #HIGH(TIME) ; =/==/==/==/==/==/==/==/==/==/==/==/==/==/==/==/==
RET
L1:    JNB   SB1, OFF_FVP ; Якщо нема пуску при ручному керуванні
ON_FVP:
SETB  FVP              ; Вмикання насоса
CLR   FVP_ALARM       ; Зняття сигналу тривоги
RET
L2:    JB    TF0, L3          ; Перехід при переповненому таймері
SETB  TR0              ; Інакше – пуск таймера 0
JMP   ON_FVP
L3:    JB    SP1, ON_FVP     ; Перевірка тиску
SETB  FVN_ALARM       ; Відімкнення за умови розгерметизації
JMP   OFF_FVP          ; форвакуумної магістралі
DN_1:  ...              ; п/п керування дифузійним насосом 1
RET
FK:    ...              ; п/п керування форвакуумним клапаном
RET
Main:  ...              ; Головний модуль системи керування
CALL  FVP              ; Форвакуумний насос
CALL  DN_1             ; Дифузійний насос 1
CALL  FK               ; Форвакуумний клапан
...
JMP   Main             ; Зациклювання головного модуля

```

Цю програму можна записати мовою AVR асемблера (дод. 1) так, як показано нижче. У цьому прикладі збережено лексичні конструкції, властиві системі команд процесора *18051*, але через відсутність в AVR-пристрої команд булевого процесора команди *JB* і *JNB*, а також *CLR* і *SETB* виконано з допомогою макровизначень, які починаються з підкреслення. Ця робота потребує додаткового клопоту, але значно спрощує написання програми:

```

.macro JB
    SBIC @0, @1
    RJMP @2
.endm
.macro JNB
    SBIS @0, @1
    RJMP @2
.endm
.macro _CLR_TF1
.endm

```

```

        .macro _CLR_TR1
        .endm
        .macro _SETB_TR1
        .endm

FVP:
    _JB PINB, Alarm, OFF_FVP      ; Перехід під час аварії
    _JB PINB, SQ1, L1             ; Перехід за наявності перемички «Test»
    _JB PINB, SA1, L1             ; Перехід на ручне керування
    _JNB PINB, SQ2, OFF_FVP       ; Перехід при відчиненій камері
    _JB PINB, SB1, L2             ; Перехід при ввімкненому автоматичному
                                ; режимі
    _JB PINB, SQT, L2             ; Невимкнення за умови високої
                                ; температури

OFF_FVP:
    CBI PORTC, FVP               ; Pump Off
    _CLR_TF1                     ; Повернення таймера у вихідний стан
    _CLR_TR1                     ; Зняття сигналу збудження таймера
    LDI Temp, #LOW(TIME)
    OUT TCNTL1, Temp             ; Перезавантаження часу затримки
    LDI Temp, #HIGH(TIME)
    OUT TCNTH1, Temp             ; /=/=/=/=/=/=/=/=/=/=/=/=/=/=/=/=
    RET
L1:    _JNB PINB, SB1, OFF_FVP ; Якщо нема пуску при ручному
                                ; керуванні

ON_FVP:
    SBI PORTC, FVP               ; Pump On
    CBI PORTC, FVP_ALRM
    RET
L2:    _JB TF1, L3               ; Перехід при переповненому таймері
    _SETB_TR1                    ; Інакше – пуск таймера 0
    RJMP ON_FVP
L3:    _JB PINB, SP1, ON_FVP     ; Перевірка тиску
    SBI PORTC, FVP_ALRM         ; Відімкнення за умови розгерметизації
    RJMP OFF_FVP                ; форвакуумної магістралі
DN_1:  ...                      ; п/п керування дифузійним насосом 1
    RET
FK:    ...                      ; п/п керування форвакуумним клапаном
    RET
Main:                                     ; Головний модуль системи керування
    RCALL FVP                   ; Форвакуумний насос
    RCALL DN_1                  ; Дифузійний насос 1
    RCALL FK                    ; Форвакуумний клапан ...
    RJMP Main                   ; Зациклювання головного модуля

```

У прикладі не розглянуто механізм призначення змінних виводам МП пристрою та робота з таймером. Приклад керування ФВН як частиною складної системи за структурним принципом визначено організацією головної програми (модуль *MAIN*), яка складається з кількох програмних модулів керування всіма об'єктами системи, об'єднаними спільним

виконанням технологічного завдання створення вакууму. Програмні модулі пов'язані між собою як сигналами стану (*on/off*, *open/closed*, *busy*, *redy*) певного обладнання, сигналами сенсорних пристроїв автоматики: рівень тиску, рівень температури, рівень струму, рівень напруги, час роботи, так і сигналами керування, які надходять з пульта оператора або керувальної програми в автоматичному режимі керування.

Керувальну програму можна записати іншою мовою, наприклад C, але так чи інакше її буде зведено до мови асемблера МП пристрою певним чином налаштованими засобами компіляції. Одним із чинників налаштування є «рівень оптимізації» керувальної програми. При певних налаштуваннях компілятор може навіть зовсім скасувати код користувача, якщо він не наполягатиме на цьому спеціальними керувальними вказівками. Програма, отримана з допомогою транслятора з мови високого рівня, виявляється значно більшою за об'ємом і довшою за часом виконання. Найшвидші й найпрозоріші програми керування ТО зазвичай написано мовою асемблера.

Однак не все так просто, коли програма має великий обсяг. Мова високого рівня суттєво скорочує запис КП. До того ж асемблер є безтиповою мовою. Байт є головним типом змінних асемблера AVR, який сам розробляє усі конструкції потрібних користувачу типів. З одного боку це не завжди зручно, але з іншого – позбавляє від необхідності перетворення одних типів змінних на інші (процедури перетворення він теж розробляє сам).

Функціональну схему МП-системи керування ФВН показано на рис. 4.4, а програму наведено в лістингу 3.

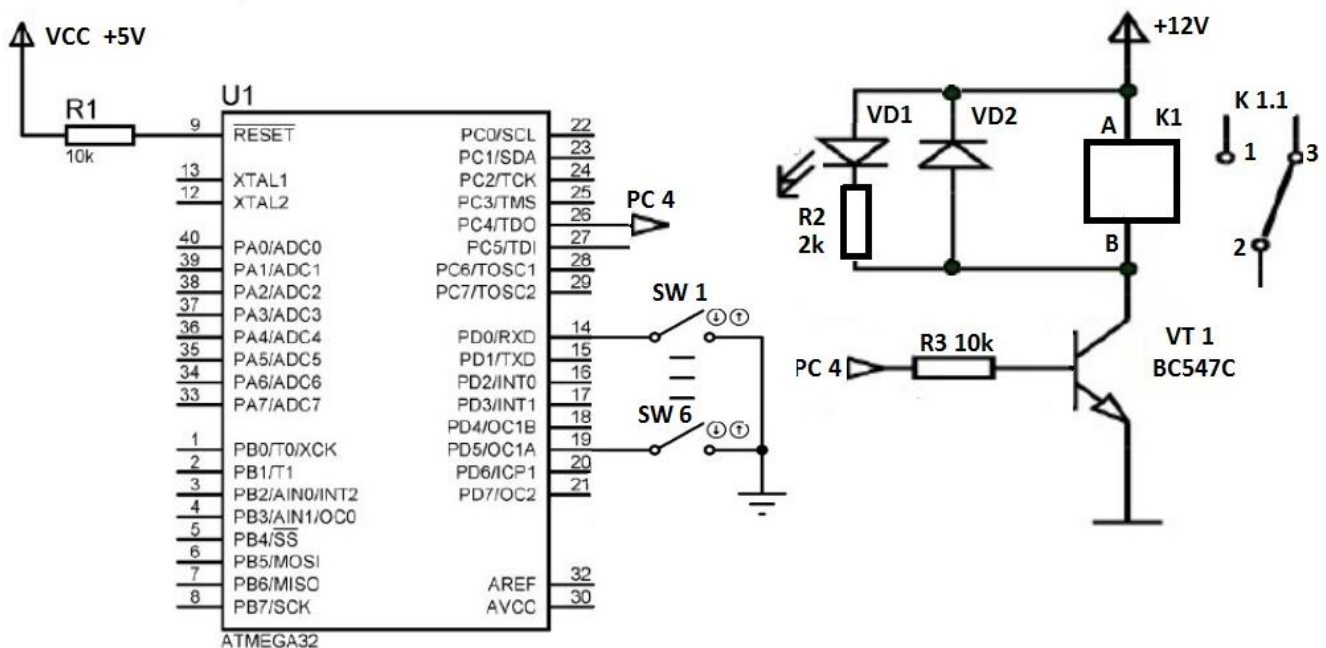


Рис. 4.4. МП-система керування пуском/зупиненням форвакуумного насоса

Лістинг 3. Програма керування форвакуумним насосом:

```
#include <avr/io.h> // FORVACUUM PUMP
//          PC4 => FVP, PC5 => ALARM
//          PD0      PD1      PD2      PD3      PD4      PD5
uint8_t  SA1=0, SB1=0, SQ1=1, SQ2=1, SP1=1, SQT=0;
uint8_t  KLAVA=0, ALRM=0;
uint16_t Time= 8000;
int temp;

int main (void)
{
    DDRC|= (1<<PC4)|(1<<PC5);
    PORTC= 0; // STOP FVP
    TCNT1L= (Time & 0xFF); // TCNT1L= LOW(Time);
    TCNT1H= (Time>>8); // TCNT1H= HIGH(Time);
    TCCR0 =0; // Timer stop TR1=0
    TIFR  =0xFF0; // TF1=0 (OVF1 reset)
    while (1)
    {
        KLAVA= PINB;
        if(PC5 ==1) goto OFF_FVP;
        else if ((PIND2 ==0)|(PIND0 ==1)) goto L1;
        else if (PIND3 ==0) goto OFF_FVP;
        else if ((PIND1 ==1)|(PIND5 ==0)) goto L2;
    OFF_FVP:
        PORTC &= !(1<<PC4); // FVP OFF
        TCCR1B=0; // Timer stop
        TCNT1L= (Time & 0xFF);
        TCNT1H= (Time>>8);
        break;
    L1: if(PIND1 ==1) goto OFF_FVP;
    ON_FVP:
        PORTC |= 1<<PC4; // FVP ON
        PORTC &= !(1<<PC5); // CLR ALARM
        break;
    L2: if ((TIFR & (1<<TOV0))!=0) goto L3;
        TCCR1B=5; // Timer start with N=1024
        goto ON_FVP;
    L3: if (PIND4 == 0) goto ON_FVP;
        PORTC |= 1<<PC5; // SET ALARM
        goto OFF_FVP;
    }
}
```

Практична робота № 8

РОЗРОБЛЕННЯ СИСТЕМИ КЕРУВАННЯ МАНІПУЛЯЦІЙНИМ РОБОТОМ

Мета роботи: вивчення методів програмування системи керування (СК) цикловим маніпуляційним роботом.

Завдання: використовуючи оболонку базової СК і її системних функцій, виконати такі дії:

- 1) налаштувати алгоритм виведення команд на виконавчі органи робота згідно з принциповою схемою МП системи;
- 2) доповнити алгоритм, наведений на рис. 4.5 програмним модулем покрокового виконання технологічного циклу робота (*STEP MODE*);
- 3) розробити систему кодування й збереження команд робота і перевірити її дію в режимі *MANUAL MODE*;
- 4) розробити підсистему виведення на дисплей режимів, коментарів і сигналів сповіщення оператора в разі невиконання команди протягом 5 с;
- 5) розробити підсистему зворотного зв'язку за сигналами стану робота;
- 6) розробити підсистему зчитування й виконання команд, що зберігаються в пам'яті програм МП, скориставшись двома підпрограмами підтримки обміну з *EEPROM*;
- 7) розробити підсистему модифікації (редагування й збереження) технологічної програми робота;
- 8) розробити підсистему дистанційного керування роботом;
- 9) розробити програмний модуль збільшення часу затримки в часі з урахуванням очікування завершення роботи пов'язаного з роботом верстата;
- 10) розробити підсистему дистанційного запису та зчитування технологічної програми;
- 11) розробити підсистему зв'язку і завантаження від ЕОМ вищого рівня;
- 12) скласти звіт про виконання роботи.

Теоретичні відомості

Згідно з принципом ієрархічності побудови системи керування [6, 7, 10] довільну (у тому числі й циклову) систему керування роботом-завантажувачем верстата можна побудувати на основі системи загального призначення, яка виконує головні функції ядра системи: керування введенням/виведенням, завданнями й задачами, пам'яттю, дисплейними функціями, зв'язком з обладнанням вищого рівня ієрархії.

Нехай робот-завантажувач виконує специфічні дії, призначені для супроводження технологічного процесу штампування, а його рухи по всіх ступенях свободи обмежено датчиками кінцевого положення захоплювального пристрою. Робот може функціонувати в тестовому або

автоматичному режимі за командами, які попередньо має бути занесено до *EEPROM*-пам'яті процесорної системи (рис. 4.5).

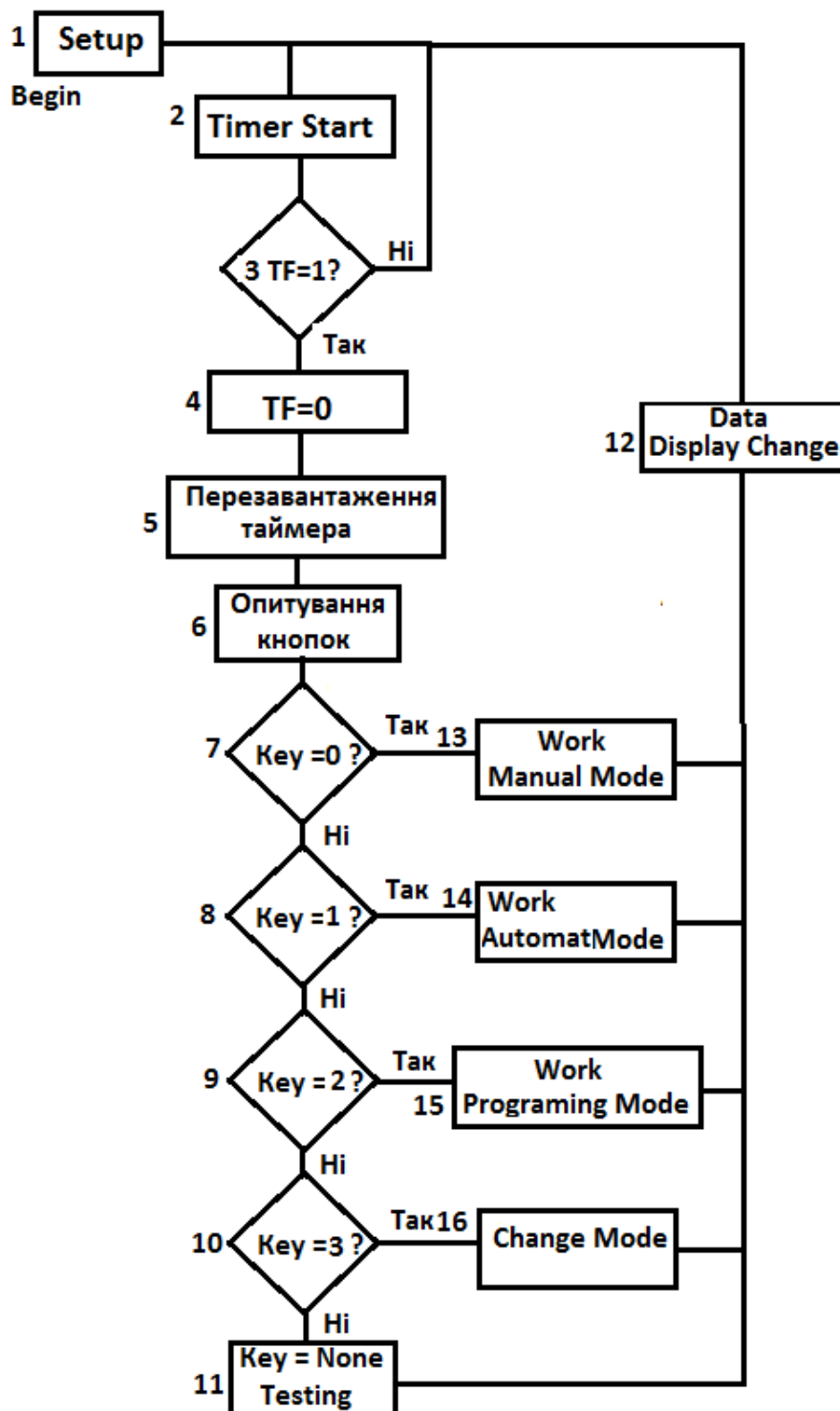


Рис. 4.5. Алгоритм роботи базової СК маніпуляційним роботом

Для проєктування програмного забезпечення системи керування необхідно розробити принципову схему МП-системи, для якої за попередніми

припущеннями можна застосувати МП-пристрій *AtMega32* родини *AVR* (32K *FLASH ROM*, 2K *RAM*, 1K *EEPROM*, *Fclk* = 0...16 МГц). Функціональну схему МП-системи показано на рис. 4.6.

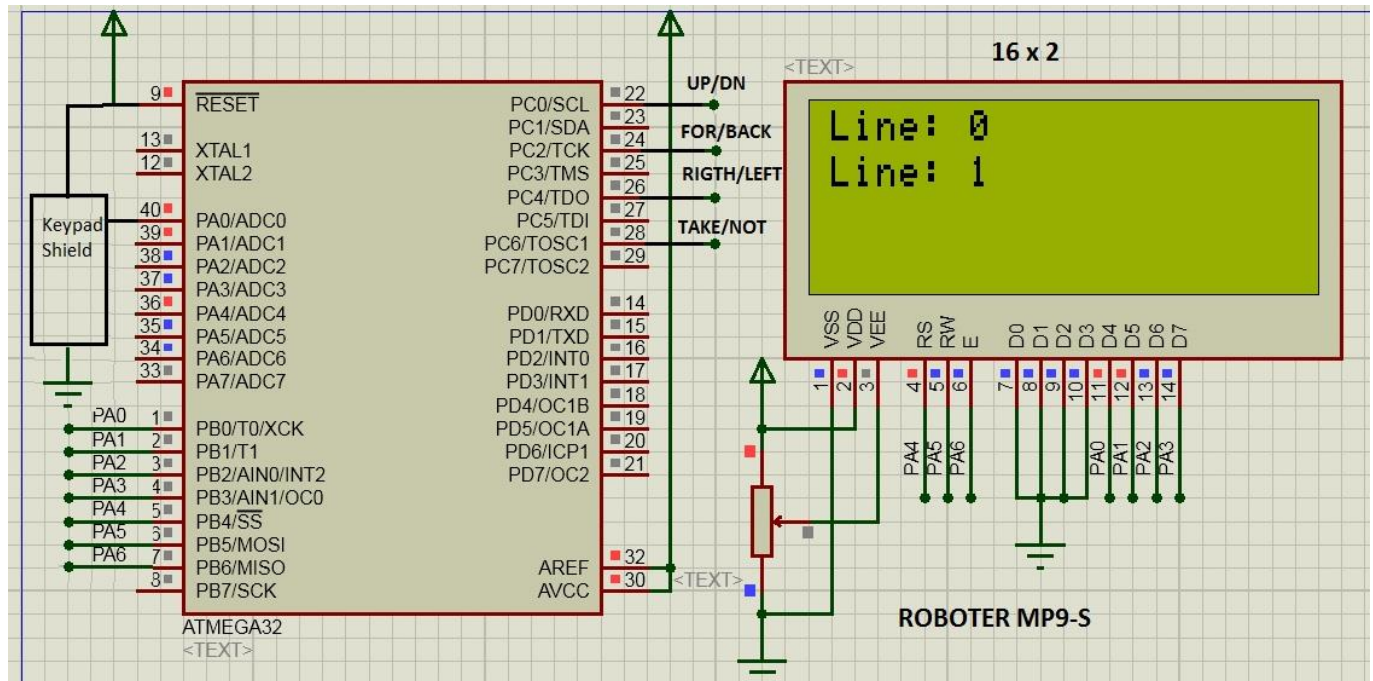


Рис. 4.6. Функціональна схема системи керування роботом-завантажувачем технологічного обладнання

На схемі виходи процесора 1–7 (PA0–PA6) призначено для передання інформації на символний дисплей системи керування типу HT 44780 (2 рядки по 16 символів); до входу 40 процесора (ADC0) під'єднано *Keypad Shield* фірми *Robotdin*, який містить шість керувальних кнопок і наведений дисплей. Виходи 22, 24, 26, 28 процесора призначено для передання керувальних сигналів на виконавчі приводи робота, призначені для виконання основних рухів за кожною з незалежних координат, а порти *PD0–PD7* процесора (виводи 14–21) призначено для введення інформації стану сенсорів зворотного зв'язку.

Лістинг 4 базової системи програми керування роботом:

```
#define F_CPU 3276800 // Частота кварцу (Гц)

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <util/delay.h>
#include <string.h>
#include <avr/eeprom.h>
// #include "lcd-library.c"
```

```

// #include <avr/pgmspace.h> for flash function
// Keyboard
#define btnRIGHT 0 // BACK/FORWARD
#define btnUP 1 // Take/No_Take
#define btnDOWN 2 // UP/DOWN
#define btnLEFT 3 // LEFT/RIGHT
#define btnSELECT 4
#define btnSTEP 5
#define btnNONE 6 // кнопки не нажаты
#define RST PD0 // ON/OFF or STEP (input, pin_14)

// Display Commands
#define RW_1 PORTB |= 4 // режим чтения
#define RS_1 PORTB |= 2 // переданная дань
#define E_1 PORTB |= 1 // строб установлено
#define RW_0 PORTB &= 3 // режим записи
#define RS_0 PORTB &= 5 // переданная команд
#define E_0 PORTB &= 6 // строба немає

// ON ROBOT
#define M_UP PORTC |= 1<<PC0 // DOWN/UP (output,
pin_22)
#define M_DN PORTC &= ~(1<<PC0)
#define M_LEFT PORTC |= 1<<PC4 // LEFT/RIGHT (output,
pin_26)
#define M_RIGHT PORTC &= ~(1<<PC4)
#define M_FORW PORTC |= 1<<PC2 // FORW/BACK (output,
pin_24)
#define M_BACK PORTC &= ~(1<<PC2)
#define M_TAKE PORTC |= 1<<PC6 // TAKE/NO_TAKE (output,
pin_28)
#define M_NO_TAKE PORTC &= ~(1<<PC6)

uint8_t COMM_EEMEM; // EEPROM DATA
uint16_t N_EEMEM; // EEPROM ADDR

uint8_t key= 1, MODE=0;
uint8_t eOn= 0, eTest= 0;

const unsigned char addLUT[4] = {0x80, 0xC0, 0x94, 0xD4};
unsigned char LCD_Address, LCD_Line;
unsigned int PreviousTime;
char buffer[15];

void WriteNibble(unsigned char data)
{
    RW_0; // WRITE OPERATOR
    E_1; // WRITE ENABLED
    PORTB = (data & 0x0F); // LOW nibbl of DATA
    E_0; // WRITE DISABLED
    RW_1; // READ OPERATOR
    _delay_loop_2(9000);

```

```

}

void Write_Byte(unsigned char data)    // запис байта в LCD
{
    RW_0;                               // Установити режим запису
    E_1;                               // Розпочати передання
    PORTB = (data >> 4);               // Передати старшу тетраду
    E_0;                               // Завершити передання
    E_1;                               // Розпочати передання
    PORTB = (data & 0xF);               // Передати молодшу тетраду
    E_0;                               // Завершити передання
    RW_1;                               // Установити режим читання
    _delay_loop_2(9840);               // Затримка в часі 3mS
}

void GoToLine(char LineNum)            // Перехід до першої позиції рядка
                                        // з номером LineNum
{
    RS_0;                               // Передання команди на LCD
    LCD_Address = addLUT[LineNum-1];    // Визначення адреси рядка
    Write_Byte(LCD_Address);            // Запис адреси рядка в LCD
    RS_1;                               // Передання команди закінчено
    LCD_Address = 0;                    // Перехід до першої позиції
    LCD_Line = LineNum;                 // Установлення поточного рядка
}

void ClearLCD(void)
{
    RS_0;                               // Передання команди на LCD
    Write_Byte(0x01);                   // Очищення RAM LCD
    _delay_loop_2(32800);               // Затримка 10 mS
    RS_1;                               // Передання команди закінчено
    GoToLine(1);                        // Перейти до нульового рядка
}

void Set_LCD_Position (char row, char col) // Установлення поточної
                                           // позиції (col)
{
    // у поточному рядку (row) LCD
    RS_0;                               // Команда на LCD модуль
    LCD_Address = addLUT[row-1] + col;   // Визначення позиції дисплея
    Write_Byte(LCD_Address);             // Запис адреси в LCD-модуль
    RS_1;                               // Передання команди закінчено
    LCD_Line = row;                      // Установлення поточного рядка
}

void Show_Char(uint8_t c)
{
    RS_1;
    Write_Byte(c);                      // Запис символу
    LCD_Address++;                       // Інкремент адреси
    switch (LCD_Address)

```

```

    {
        case 0x10: GoToLine(2); break;
        case 0x40: GoToLine(1); break;
    }
}

void ShowStr(char *s)
{
    while (*s != 0) Show_Char(*s++);
}

void InitLCD(void)
{
    RW_1;
    E_0;
    RS_0;
    _delay_loop_2(50000);
    WriteNibble(0x33);
    WriteNibble(0x33);
    WriteNibble(0x33);
    WriteNibble(0x22);
    Write_Byte(0x28);
    Write_Byte(0x01);
    Write_Byte(0x10);
    Write_Byte(0x06);
    Write_Byte(0x0C);
    for (char i=0x40; i<0x5F; i++)
    {
        _delay_loop_2(10000);
        RS_0;
        Write_Byte(i);
        _delay_loop_2(10000);
        Show_Char(0);
    }
    RS_1;
    Set_LCD_Position(1, 2);
    buffer[0] = 'R';
    buffer[1] = 'O';
    buffer[2] = 'B';
    buffer[3] = 'O';
    buffer[4] = 'T';
    buffer[5] = 'E';
    buffer[6] = 'R';
    buffer[7] = ' ';
    buffer[8] = 'M';
    buffer[9] = 'P';
    buffer[10] = '9';
    buffer[11] = 'S';
    buffer[12] = '\0';
    ShowStr(buffer);
}

```

```

/*int INTERRUPT (SIG_OWERFLOW1) // П/п оброблення переривання від Т/С1
за переповненням

{
    TCNT1 = -640;                // 3200/640    f = 20 Hz
    PORTC ^= 0x02;               // Частота звернень до АЦП – 20 Hz
    _delay_loop_2(1400);         // Закінчення АЦП
    // int adc = ADCW;           // analogRead(A0);
} */

void WORK(uint8_t K)             // Виконання команди роботом
{PORTC |=1<<K;}

void keypadAction() {
    uint8_t N_max=30;
    uint8_t N;

    switch (key)                 // Аналіз коду натиснутої кнопки
    {
        case btnRIGHT:
            if (MODE==0 || MODE==3) { if ((PORTC && (1<<PC2))!=0) M_BACK;
// Forward/Backward
                                else M_FORW; }
            if (MODE==1) while(eOn) {} // EEPROM.read(N,COMM); WORK(COMM);
// if (N<N_max && COMM!=0) N++;
                                // else eOn=0; }
            if (MODE==2) if (eOn) { } // EEPROM.read(N,COMM); WORK(COMM);
N++;
// if (COMM==0) eOn=0;}
            break;
        case btnLEFT:
            if (MODE==0 || MODE==3) { if ((PORTC && (1<<PC4))!=0) M_RIGTH; //
Left/Rigth
                                else M_LEFT; }
            if (MODE==1) while(eOn) {COMM = eeprom_read_byte(&N); if (N<N_max
&& COMM!=0) {WORK(COMM);N++;}
                                else eOn=0; }
            if ((MODE==2) && (eOn)) {COMM = eeprom_read_byte(&N); if (N<N_max
&& COMM!=0) {WORK(COMM);N++;}
                                else eOn=0; } //
EEPROM.read(N,COMM); WORK(COMM); N++;}
            break;
        case btnUP:
            if (MODE==0 || MODE==3) { if ((PORTC && (1<<PC6))!=0) M_NO_TAKE;
//Take/No_Take
                                else M_TAKE; } // лічильний тригер//
EEPROM.write(2, eOn);
            if (MODE==1) while(eOn) {COMM = eeprom_read_byte(&N);
                                if (N<N_max &&
COMM!=0) {WORK(COMM);N++;}
                                else eOn=0; }

```

```

        if ((MODE==2) && eOn) {COMM = eeprom_read_byte(&N);
                                if (N<N_max && COMM!=0) {WORK(COMM);N++;}
                                else eOn=0; }// EEPROM.read(N,COMM);
WORK(COMM); N++;}
    if (eOn) {
        Set_LCD_Position(0, 0); // lcd.print(" <<
MODE >> ");
        buffer[0] = '<';
        buffer[1] = '<';
        buffer[2] = 'M';
        buffer[3] = 'O';
        buffer[4] = 'D';
        buffer[5] = 'E';
        buffer[6] = MODE;
        buffer[7] = '\0';
        ShowStr(buffer);
        // sprintf(buffer," %02d C", MODE); // lcd.print(" << COMMAND >>
        ");
        Set_LCD_Position(1, 0);
        buffer[0] = '>';
        buffer[1] = '>';
        buffer[2] = 'C';
        buffer[3] = 'O';
        buffer[4] = 'M';
        buffer[5] = 'M';
        buffer[6] = COMM;
        buffer[7] = '\0';
        ShowStr(buffer);
        // sprintf(buffer," %02d C", (uint8_t)N); // Step (номер команды)
        }
        break;
    }
    case btnDOWN:
    { if (MODE==0 || MODE==3) { if ((PORTC && (1<<PC0))==0) M_DN; //
UP/DOWN
        else M_UP; }
        if (MODE==1) while(eOn) {} // EEPROM.read(N,COMM); WORK(COMM);
// if (N<N_max && COMM!=0) N++; else eOn=0; }
        }
        break;
    }
    case btnSELECT: // MODE Selector
    { MODE++; // Change MODE
        if (MODE>3) {eOn=0; MODE=0;}
        // 0 - MANUAL MODE
        // 1 - AUTOMAT MODE
        // 2 - CIKLE MODE
        // 3 - TEST MODE
        // EEPROM.write(1, MODE);
    }
    break;
    case btnSTEP: // On/Off or Step Selector

```

```

        { eOn ^= eOn; PORTB ^= (1<<PB7)
        }
        break;
    case btnNONE:
        {Set_LCD_Position(0, 0);                // lcd.print("  <<  MODE >>
");
        // sprintf(buffer," %02d C", MODE);        // lcd.print("  <<
COMMAND >>  ");
            buffer[0] = 'M';
            buffer[1] = 'O';
            buffer[2] = 'D';
            buffer[3] = 'E';
            buffer[4] = MODE;
            buffer[5] = '\\0';
        ShowStr(buffer);
        Set_LCD_Position(1, 0);
            buffer[0] = 'C';
            buffer[1] = 'O';
            buffer[2] = 'M';
            buffer[3] = 'M';
            buffer[4] = COMM;
            buffer[5] = '\\0';
        ShowStr(buffer);
        // sprintf(buffer," %02d C", (uint8_t)COMM);
        }
        break; // no key is pressed; quit
        // Set_LCD_Position(4, 8);
        // ShowStr(buffer);
        }
    }

```

```

uint8_t EEPROM_ReadByte(uint8_t *adr)
{
    while (EECR & (1<<EWE));
    EEAR = (uint16_t)adr;
    EECR |= (1<<EERE);
    return EEDR;
}
/*

```

Приклад використання:

```

uint8_t data EEMEM;
uint8_t tmp; // &data => ADDRESS
tmp = eeprom_read_byte(&data);
eeprom_write_byte(&data, 125); */

```

```

void EEPROM_WritByte(uint8_t *adr, uint8_t value)
{
    while (EECR & (1<<EWE));
    EEAR = (uint16_t)adr;
    EEDR =value;
    EECR |= (1<<EEMWE);
    EECR |= (1<<EWE);
}

```

```

}

int main (void)
{
    DDRA = 0;
    DDRC |= (1<<0) | (1<<2) | (1<<4) | (1<<6);
    DDRB = 0xFF;
    DDRD = 0; //On Robot 6-TAKE, 5-F/B, 4-L/R, 3-U/D, 2-RST,
1- BLINKER
    // TIMSK = _BV(TOIE1); // TIM1 int ENABLED! TCCR1A
=0 // TCCR1B = _BV(CS11);
    TCNT1 = -100;
    TCCR1B = _BV(CS12); // START TIMER1 with Fclk/1024
    ACSR = _BV(ACD); // Відмикання аналогового компаратора
    ADMUX = 0; // KLAVA // ADFR __BV(ADIE) |
_BV(ADSC)
    ADCSRA = _BV(ADEN) | _BV(ADATE) | _BV(ADPS2) | _BV(ADPS1); // f/64
    //SFIOR |= 1<<ADTS2 | 1<<ADTS1; // Старт АЦП за переповненням
таймера T1
    InitLCD();
    TIFR |= (1<<TOV1); // Скидання прапорця переповнення
таймера T1
    cli (); // sei ();
    if ((PINB & (1<<PB7))==0) eOn=0;
    else eOn=1;
    while(1)
    {
        while((TIFR & (1<<TOV1))==0);
        {
            TIFR |= (1<<TOV1); // Скидання прапорця переповнення таймера
T1
            ADCSRA |= 1<< ADIF; // Скидання прапорця завершення АЦП
            ADCSRA |= 1<< ADSC; // START ADC
while ((ADCSRA & (1<< ADIF))==0); // Очікування завершення АЦП
            // TIFR |= (1<<TOV1); // Скидання прапорця переповнення
таймера T1
            TCCR1B = _BV(CS12); // Установлення прескалера таймера T1
            TCNT1 = -640; // 3200/640 => 20 Hz (установлення
затримки T1)
            PORTD ^= 0x01; // Індикатор звернень до АЦП F=20 Hz
        }

        int adc = ADCW; // analogRead(A0);
        if ((adc < 50) && (adc>0)) {key= btnRIGHT; goto Exit_;;}
        //0
        if ((adc < 200) && (adc>50)) {key= btnUP; goto Exit_;;}
        //1
        if ((adc < 300) && (adc>200)) {key= btnDOWN; goto Exit_;;} //2
        if ((adc < 500) && (adc>300)) {key= btnLEFT; goto Exit_;;}
        //3
        if ((adc < 700) && (adc>500)) {key= btnSTEP; goto Exit_;;}
        //4

```

```

if ((adc < 850) && (adc>700)) {key= btnSELECT;goto Exit_;;} //5
//if (key not pressed) key= btnNONE;
Exit_::
keypadAction();
};
}

```

Нагадаємо, що в середовищі *AVRStudio* всі компоненти програмного забезпечення доведеться розробляти на власний розсуд, користуючись відповідними документами фірм-виробників як цифрового обладнання (дисплеїв, перетворювачів даних, мережних контролерів, сенсорних пристроїв тощо), так і посібників з програмування в середовищі *GNU CC*. (Саме цей різновид мови C використовується в середовищі *AVRStudio*.) Так, для підтримки символьного двострокового дисплея типу HT 44780 використано бібліотеку функцій, яку запозичено з [4], для програмної підтримки таймера й аналого/цифрового перетворювача використано рекомендації, наведені на сайті виробника [www.Atmel.com/]. Для підтримки пристрою *KeyPad* (фірми *Robotdin*) використано його принципову електричну схему, яку попередньо незначно модифіковано [www.robotdin.com].

Алгоритм роботи системи керування, показаний на рис. 4.5, складається з таких частин: модуль 1 (*Setup*), призначений для початкової ініціалізації МП системи (порти, таймери, режими обміну даними) до робочого стану (*Manual Mode*), захват справа, ззаду, знизу, розжатий. Ініціалізація й запуск таймера 2 приводить до періодичної з інтервалом 5 мс перевірки стану клавіатури (блок 6), що відбувається після появи прапорця переповнення таймера, на яке припадає очікування в блоці 3. Після переповнення відбувається програмне скидання цього прапорця у блоці 4 та перезавантаження таймера для роботи в новому циклі 5. У блоках 7–10 аналізується код натиснутої кнопки й здійснюється перехід до певних дій системи в блоках 11–16. Якщо не натиснуто жодної з кнопок, то система виконує самотестування й переходить до блока відображення режимів на дисплеї СК 16, залишаючись у тому самому режимі роботи, який зберігається у змінній «*MODE*». У блоках 13–15 аналізуються й виконуються надані команди: у блоці 13 – з кнопкового терміналу, у блоці 14 – з пам'яті програм (*EEPROM CPU*). У блоці 16 відбувається інкремент змінної «*MODE*», що змінює режим роботи СК. Максимальна кількість режимів – чотири (0, 1, 2, 3).

Базові системні функції (підтримка таймера, дисплея та клавіатури) налагоджено, їх можна використовувати під час подальшого розроблення СК.

ПРИКЛАД ПРОГРАМИ ПІДТРИМКИ КВАДРАТУРНОГО ЕНКОДЕРА BE178-A5 У СЕРЕДОВИЩІ ARDUINO-IDE МОВОЮ C-СКЕТЧ ARDUINO

```

/*
** Optical Incremental Encoder BE178-A5    (EC11 or Sparkfun)
*/
// 14500000
volatile unsigned long brightness = 14500000; // DIAPAZON (Fmax*10
Hz)
int fadeAmount ;    // Step Change
unsigned long currentTime;
unsigned long loopTime=0;
const int LED      = 13;        // d2=int0 d3=int1
const int pin_A    = 12;        // pin 12
const int pin_B    = 11;        // pin 11
const int pin_S    = 10;        // pin 10 Step

unsigned char encoder_A;
unsigned char encoder_B;
unsigned char encoder_S;
unsigned char encoder_A_prev=0;
unsigned char encoder_B_prev=0;
unsigned char encoder_S_prev=0;

#include "Wire.h"

void setup()
{ pinMode(9, OUTPUT); // declare pin 9 to be an output:
  pinMode(13,OUTPUT);
  pinMode(pin_A, INPUT); // declare pins 10..12 as inputs
  pinMode(pin_B, INPUT);
  pinMode(pin_S, INPUT);
  currentTime = millis();
  loopTime = currentTime;
  Serial.begin(115200);
  attachInterrupt(0, Sens_R, RISING);
  attachInterrupt(1, Sens_L, RISING);
  pinMode(2, INPUT);
  pinMode(3, INPUT);
}
void Sens_L()
{
  if (digitalRead(2)) {brightness=brightness+1; }
  Serial.println(brightness);
  // виводимо показчики
}
void Sens_R()
{
  if (digitalRead(3)) {brightness=brightness-1;
  Serial.println(brightness);} // виводимо показчики }
void loop() { }

```

КОМАНДИ АСЕМБЛЕРА

Таблиця Д.2.1

Команди асемблера процесорів родини AVR

Мнемоніка	Опис	Операція	Цикли	Прапорці
AND Rd, Rr	«Логічне І» двох РЗП	$Rd \leftarrow RD \cdot Rr$	1	Z,N,V
ANDI Rd,K	«Логічне І» РЗП і константи	$Rd \leftarrow RD \cdot K$	1	Z,N,V
EOR Rd, Rr	«Виключне АБО» двох РЗП двох	$Rd \leftarrow RD \oplus Rr$	1	Z,N,V
OR	«Логічне АБО» двох РЗП	$Rd \leftarrow RD + Rr$	1	Z,N,V
ORI Rd, K	«Логічне АБО» РЗП і константи	$Rd \leftarrow RD \vee K$	1	Z,N,V
COM Rd	Перетворення на зворотний код	$Rd \leftarrow 0FFH - Rd$	1	Z,C,N,V
NEG Rd	Перетворення на додатковий код	$Rd \leftarrow 00H - Rd$	1	Z,C,N,V,H
CLR	Переведення у додатковий код	$Rd \leftarrow 0$	1	Z,N,V
SER Rd	Установлення всіх розрядів РЗП	$Rd \leftarrow 0FFH$	1	–
TST	Перевірка РЗП на від'ємне (нульове) значення	$Rd \leftarrow RD \cdot RD$	1	Z,N,V

Примітка. РЗП – регістр загального призначення, ОЗП – оперативний запам'ятовувальний пристрій.

Таблиця Д.2.2

Група команд арифметичних операцій

Мнемоніка	Опис	Операція	Цикли	Прапорці
ADD Rd, Rr	Додавання двох РЗП	$Rd \leftarrow Rd + Rr$	1	Z, C, N, V, H
ADC Rd, Rr	Додавання двох РЗП з перенесенням	$Rd \leftarrow Rd + Rr + c$	1	Z, C, N, V, H
ADIW Rd, K	Складання регістрової пари з константою	$Rdh:Rdl \leftarrow Rdh : Rdl + K$	2	Z, C, N, V, S
SUB Rd, Rr	Віднімання двох РЗП	$Rd \leftarrow Rd - Rr$	1	Z, C, N, V, H
SUBI Rd, K	Віднімання константи з РЗП	$Rd \leftarrow Rd - K$	1	Z, C, N, V, H
SBC Rd, Rr	Віднімання двох РЗП із запозиченням	$Rd \leftarrow Rd - Rr - C$	1	Z, C, N, V, H
SBCI Rd, K	Віднімання константи з РЗП з урахуванням перенесення	$Rd \leftarrow Rd - K - C$	1	Z, C, N, V, H
SBIW Rd, K	Віднімання константи з регістрової пари	$Rdh : Rdl \leftarrow Rdh : Rdl - K$	2	Z, C, N, V, S
DEC Rd	Декремент РЗП	$Rd \leftarrow Rd - 1$	1	Z,N,V
INC Rd	Інкремент РЗП	$Rd \leftarrow Rd + 1$	1	Z,N,V
CP Rd, Rr	Порівняння двох РЗП	$?(Rd - Rr)$	1	Z, C, N, V, H
CPC Rd, Rr	Порівняння двох РЗП з урахуванням перенесення	$?(Rd - Rr - C)$	1	Z, C, N, V, H
CPI Rd, K	Порівняння РЗП із константою	$?(Rd - K)$	1	Z, C, N, V, H

Група команд операцій з розрядами

Мнемоніка	Опис	Операція	Цикли	Прапорці
CBR Rd, K	Скидання розряду(ів) P3П	$Rd \leftarrow Rd \cdot (0FFH - K)$	1	Z, N, V
SBR Rd, K	Установлення розряду(ів) P3П	$Rd \leftarrow Rd \vee K$	1	Z, N, V
CBI A, Ь	Скидання розряду PBB	$A.Ь \leftarrow 0$	2	–
SBI A, Ь	Установлення розряду PBB	$A.Ь \leftarrow 1$	2	–
BCLR s	Скидання прапорця	$SREG.s \leftarrow 0$	1	SREG.s
BSET s	Установлення прапорця	$SREG.s \leftarrow 1$	1	SREG.s
BLD Rd, Ь	Завантаження розряду P3П із прапорця T	$Rd.Ь \leftarrow T$	1	–
BST Rd, Ь	Запис розряду P3П у прапорець T (SREG)	$T \leftarrow Rd.Ь$	1	T
CLC	Скидання прапорця перенесення	$C \leftarrow 0$	1	C
SEC	Установлення прапорця перенесення	$C \leftarrow 1$	1	C
CLN	Скидання прапорця негативної кількості	$N \leftarrow 0$	1	N
SEN	Установлення прапорця негативної кількості	$N \leftarrow 1$	1	N
CLZ	Скидання прапорця нуля	$Z \leftarrow 0$	1	Z
SEZ	Установлення прапорця нуля	$Z \leftarrow 1$	1	Z
CLI	Загальна заборона переривань	$I \leftarrow 0$	1	I
SEI	Загальний дозвіл переривань	$I \leftarrow 1$	1	I
CLS	Скидання прапорця знака	$S \leftarrow 0$	1	S
SES	Установлення прапорця знака	$S \leftarrow 1$	1	S
CLV	Скидання прапорця переповнення додаткового коду	$V \leftarrow 0$	1	V
SEV	Установлення прапорця переповнення додаткового коду	$V \leftarrow 1$	1	V
CLT	Скидання прапорця T	$T \leftarrow 0$	1	T
SET	Установлення прапорця користувача T	$T \leftarrow 1$	1	T
CLH	Скидання прапорця половинного перенесення	$H \leftarrow 0$	1	H
SEH	Установлення прапорця половинного перенесення	$H \leftarrow 1$	1	H

Таблиця Д.2.4

Група команд операцій запису/читання

Мнемоніка	Опис	Операція	Цикли	Прапорці
MOV Rd, Rr	Пересилання між РЗП	$Rd \leftarrow Rr$	1	–
MOVW	Пересилання між парами регістрів	$Rd + 1:Rd \leftarrow Rr + 1, Rr$	1	–
LDI Rd, K	Завантаження константи в РЗП	$Rd \leftarrow K$	1	–
LD Rd, X	Непряме читання	$Rd \leftarrow [X]$	2	–
LD Rd, X+	Непряме читання з постінкрементом	$Rd \leftarrow [X], X \leftarrow X + 1$	2	–
LD Rd, -X	Непряме читання з переддекрементом	$X \leftarrow X - 1, Rd \leftarrow [X]$	2	–
LD Rd, Y	Непряме читання	$Rd \leftarrow [Y]$	2	–
LD Rd, Y+	Непряме читання з постінкрементом	$Rd \leftarrow [Y], Y \leftarrow Y + 1$	2	–
LD Rd, -Y	Непряме читання з переддекрементом	$Y \leftarrow Y - 1, Rd \leftarrow [Y]$	2	–
LD Rd, Z+q	Непряме відносне читання	$Rd \leftarrow [Y + q]$	2	–
LD Rd, Z	Непряме читання	$Rd \leftarrow [Z]$	2	–
LD Rd, Z+	Непряме читання з постінкрементом	$Rd \leftarrow [Z], Z \leftarrow Z + 1$	2	–
LD Rd, -Z	Непряме читання з переддекрементом	$Z \leftarrow Z - 1, Rd \leftarrow [Z]$	2	–
LD Rd, Z+q	Непряме відносне читання	$Rd \leftarrow [Z + q]$	2	–
LDS Rd, K	Безпосереднє читання з ОЗП	$Rd \leftarrow [K]$	2	–
ST X, Rr	Непрямий запис	$[X] \leftarrow Rr$	2	–
ST X+, Rr	Непрямий запис із постінкрементом	$[X] \leftarrow Rr, X \leftarrow X + 1$	2	–
ST -X, Rr	Непрямий запис із переддекрементом	$X \leftarrow X - 1, [X] \leftarrow Rr$	2	–
ST Y, Rr	Непрямий запис	$[Y] \leftarrow Rr$	2	–
ST Y+, Rr	Непрямий запис із постінкрементом	$[Y] \leftarrow Rr, Y \leftarrow Y + 1$	2	–
ST -Y, Rr	Непрямий запис із переддекрементом	$Y \leftarrow Y - 1, [Y] \leftarrow Rr$	2	–
ST Y+q, Rr	Непрямий відносний запис	$[Y + q] \leftarrow Rr$	2	–
ST Z, Rr	Непрямий запис	$[Z] \leftarrow Rr$	2	–
ST Z+, Rr	Непрямий запис із постінкрементом	$[Z] \leftarrow Rr, Z \leftarrow Z + 1$	2	–
ST -Z, Rr	Непрямий запис із переддекрементом	$Z \leftarrow Z - 1, [Z] \leftarrow Rr$	2	-

Продовження табл. Д.2.4

Мнемоніка	Опис	Операція	Цикли	Прапорці
ST Z+q,Rr	Непрямий відносний запис	$[Z+q] \leftarrow Rr$	2	–
STS k, Rr	Безпосередній запис в ОЗП	$[k] \leftarrow Rr$	2	–
LPM	Завантаження даних із 3П програм	$Rd \leftarrow \{Z\}$	3	–
LPM Rd, Z	Завантаження даних із 3П програм	$Rd \leftarrow \{Z\}$	3	–
LPM Rd,Z+	Завантаження даних із 3П програм і постінкремент	$Rd \leftarrow \{Z\}, Z \leftarrow Z + 1$	3	–
MOV Rd, Rr	Пересилання між РЗП	$Rd \leftarrow Rr$	1	–
MOVW	Пересилання між парами регістрів	$Rd + 1:Rd \leftarrow Rr + 1, Rr$	1	–
LDI Rd, K	Завантаження константи в РЗП	$Rd \leftarrow K$	1	–
LD Rd, X	Непряме читання	$Rd \leftarrow [X]$	2	–
LD Rd, X+	Непряме читання з постінкрементом	$Rd \leftarrow [X], X \leftarrow X + 1$	2	–
LD Rd,-X	Непряме читання з переддекрементом	$X \leftarrow X - 1, Rd \leftarrow [X]$	2	–
LD Rd,Y	Непряме читання	$Rd \leftarrow [Y]$	2	–
LD Rd, Y+	Непряме читання з постінкрементом	$Rd \leftarrow [Y], Y \leftarrow Y + 1$	2	–
LD Rd, -Y	Непряме читання з переддекрементом	$Y \leftarrow Y - 1, Rd \leftarrow [Y]$	2	–
LD Rd, Z+q	Непряме відносне читання	$Rd \leftarrow [Y + q]$	2	–
LD Rd,Z	Непряме читання	$Rd \leftarrow [Z]$	2	–
LD Rd, Z+	Непряме читання з постінкрементом	$Rd \leftarrow [Z], Z \leftarrow Z + 1$	2	–
LD Rd, -Z	Непряме читання з переддекрементом	$Z \leftarrow Z - 1, Rd \leftarrow [Z]$	2	–
LD Rd, Z+q	Непряме читання з переддекрементом	$Rd \leftarrow [Z + q]$	2	–
LDS Rd, K	Безпосереднє читання із ОЗП	$Rd \leftarrow [K]$	2	–
ST X, Rr	Непрямий запис	$[X] \leftarrow Rr$	2	–
ST X+, Rr	Непрямий запис із постінкрементом	$[X] \leftarrow Rr, X \leftarrow X + 1$	2	–
ST -X, Rr	Непрямий запис із переддекрементом	$X \leftarrow X - 1, [X] \leftarrow Rr$	2	–
ST Y, Rr	Непрямий запис	$[Y] \leftarrow Rr$	2	–
ST Y+, Rr	Непрямий запис із постінкрементом	$[Y] \leftarrow Rr, Y \leftarrow Y + 1$	2	–

Мнемоніка	Опис	Операція	Цикли	Прапорці
ST -Y, Rr	Непрямий запис із переддекрементом	$Y \leftarrow Y - 1, [Y] \leftarrow Rr$	2	–
ST Y+q, Rr	Непрямий відносний запис	$[Y + q] \leftarrow Rr$	2	–
ST Z, Rr	Непрямий запис	$[Z] \leftarrow Rr$	2	–
ST Z+, Rr	Непрямий запис із постінкрементом	$[Z] \leftarrow Rr, Z \leftarrow Z + 1$	2	–
ST -Z, Rr	Непрямий запис із переддекрементом	$Z \leftarrow Z - 1, [Z] \leftarrow Rr$	2	–
ST Z+q,Rr	Непрямий відносний запис	$[Z + q] \leftarrow Rr$	2	–
STS k, Rr	Непрямий відносний запис в ОЗП	$[k] \leftarrow Rr$	2	–
LPM	Завантаження даних із ЗП програм	$Rd \leftarrow \{Z\}$	3	–
LPM Rd, Z	Завантаження даних із ЗП програм	$Rd \leftarrow \{Z\}$	3	–
LPM Rd,Z+	Завантаження даних із ЗП програм і постінкремент	$Rd \leftarrow \{Z\}, Z \leftarrow Z + 1$	3	–

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Horowitz, P. The art of electronics V. 1 / P. Horowitz, W. Hill. – Cambridge : University Press, 1980. – 598 p.
2. Titze, U. Halbleiter Schaltungstechnik / U. Titze, P. Schenk. – Berlin : Springer Verlag, 1980. – 512 p.
3. Sommer, U. Arduino Mikrocontroller-Programmierung mit ARDUINO/FREEDUINO / U. Sommer. – Haar : FRANZIS' Verlag GmbH, 2010. – 256 p.
4. Шпак, Ю. О. Програмування на мові С для AVR та PIC мікроконтролерів / Ю. О. Шпак. – Київ : МК Пресс, 2006. – 400 с.
5. Інтегроване комп'ютерне середовище програмування «Owen Logic» (OnLine help) [Електронний ресурс]. – Режим доступу : <https://owen.ua/ru/programmnoe-obespechenie/programmnoe-obespechenie-owen-logic/>
6. ПЛК ПР-200 фірми «Овен». Настанова щодо експлуатації [Електронний ресурс]. – Режим доступу : http://owen.ua/uploads/130/re_oven_sp3hh_2-uk-94490-1.4_a4.pdf.
7. Технічні засоби автоматизації. У 2 ч. Ч. 2 / М. В. Лукінюк, В. П. Лисенко, В. Є. Лукін та ін. – Ніжин : Вид. Лисенко М. М., 2018. – 455 с.

ЗМІСТ

Вступ.....	3
1. Елементи теорії дискретних автоматів.....	5
1.1. Логічні функції та методи їх подання.....	5
1.2. Аксиоми й постулати булевої алгебри.....	8
1.3. Синтез дискретного автомата.....	12
1.4. Синтез комбінаційних логічних схем.....	13
2. Основи програмування мовою С.....	19
2.1. Структура програми мовою С.....	21
2.2. Типи даних, змінні, константи.....	22
2.3. Правила перетворення даних однієї системи числення на іншу...	23
3. Цифрові пристрої автоматики.....	48
Практична робота № 1. Вивчення роботи базових логічних елементів структури ТТЛ.....	48
Практична робота № 2. Розроблення цифрового керувального автомата.....	53
Практична робота № 3. Аналого-цифрове перетворення даних	66
Практична робота № 4. Растрові фазові перетворювачі переміщень ФПП з підрахунком муарових смуг (накопичувального типу)	79
4. Проектування мікропроцесорних систем автоматики	83
Практична робота № 5. Програмування портів увведення/виведення мікропроцесора.....	83
Практична робота № 6. Програмування часових інтервалів імпульсних сигналів.....	85
Практична робота № 7. Програмування МП керувального автомата	88
Практична робота № 8. Розроблення системи керування маніпуляційним роботом.....	95
Додаток 1. Приклад програми підтримки квадратурного енкодера BE178-A5 у середовищі ARDUINO-IDE мовою С-скетч ARDUINO.....	106
Додаток 2. Команди асемблера	107
Бібліографічний список	112

Навчальне видання

**Белявський Олександр Вадимович
Косенко Оксана Валеріївна
Бреус Андрій Олександрович**

**ОСНОВИ МІКРОПРОЦЕСОРНОЇ ТЕХНІКИ
ТА ЇЇ ПРОГРАМУВАННЯ МОВОЮ С**

Редактор О. Ф. Серьожкіна

Зв. план, 2025

Підписано до видання 25.11.2025

Ум. друк. арк. 6,3. Обл.-вид. арк. 7,13. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет
«Харківський авіаційний інститут»
61070, Харків-70, вул. Вадима Манька, 17
[http:// www.khai.edu](http://www.khai.edu)

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001