МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

УТВЕРЖДАЮ
Директор ИЭиАС
_____ С.И. Лукьянов

26.02.2020 г.

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ (МОДУЛЯ)

ОСНОВЫ МИКРОПРОЦЕССОРНОЙ ТЕХНИКИ

Направление подготовки (специальность)
13.03.02 Электроэнергетика и электротехника

Направленность (профиль/специализация) программы
Электропривод и автоматика

Уровень высшего образования - бакалавриат

Форма обучения
заочная

Институт/ факультет	Институт энергетики и автоматизированных систем
Кафедра	Автоматизированного электропривода и мехатроники
Курс	4

Магнитогорск
2019 год

Рабочая программа составлена на основе ФГОС ВО по направлению подготовки 13.03.02 Электроэнергетика и электротехника (уровень бакалавриата) (приказ Минобрнауки России от 28.02.2018 г. № 144)

Рабочая программа рассмотрена и одобрена на заседании кафедры Автоматизированного электропривода и мехатроники
13.02.2020, протокол № 6

Зав. кафедрой _____ А.А. Николаев

Рабочая программа одобрена методической комиссией ИЭиАС
26.02.2020 г. протокол № 5

Председатель _____ С.И. Лукьянов

Рабочая программа составлена:
доцент кафедры АЭПиМ, канд. техн. наук _____ О.С. Малахов

Рецензент:
зам. начальника ЦЭТЛ ПАО «ММК» по электроприводу, канд. техн. наук

_____ А.Ю. Юдин

Лист актуализации рабочей программы

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2020 - 2021 учебном году на заседании кафедры Автоматизированного электропривода и мехатроники

Протокол от 30 08 2020 г. № 1
Зав. кафедрой _____ А.А. Николаев

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2021 - 2022 учебном году на заседании кафедры Автоматизированного электропривода и мехатроники

Протокол от __ _____ 20__ г. № __
Зав. кафедрой _____ А.А. Николаев

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2022 - 2023 учебном году на заседании кафедры Автоматизированного электропривода и мехатроники

Протокол от __ _____ 20__ г. № __
Зав. кафедрой _____ А.А. Николаев

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2023 - 2024 учебном году на заседании кафедры Автоматизированного электропривода и мехатроники

Протокол от __ _____ 20__ г. № __
Зав. кафедрой _____ А.А. Николаев

1 Цели освоения дисциплины (модуля)

Целями освоения дисциплины (модуля) «Основы микропроцессорной техники» является развитие у студентов личностных качеств, а также формирование профессиональных компетенций в соответствии с требованиями ФГОС ВПО по направлению 130302 Электроэнергетика и электротехника.

2 Место дисциплины (модуля) в структуре образовательной программы

Дисциплина Основы микропроцессорной техники входит в часть учебного плана формируемую участниками образовательных отношений образовательной программы.

Для изучения дисциплины необходимы знания (умения, владения), сформированные в результате изучения дисциплин/ практик:

Схемотехника

Алгебра логики и основы дискретной техники

Знания (умения, владения), полученные при изучении данной дисциплины будут необходимы для изучения дисциплин/практик:

Системы управления электроприводов

3 Компетенции обучающегося, формируемые в результате освоения дисциплины (модуля) и планируемые результаты обучения

В результате освоения дисциплины (модуля) «Основы микропроцессорной техники» обучающийся должен обладать следующими компетенциями:

Код индикатора	Индикатор достижения компетенции
ПК-2 Способность подготовить техническое задание на разработку системы электропривода	
ПК-2.1	Осуществляет подготовку технического задания на разработку системы электропривода

4. Структура, объём и содержание дисциплины (модуля)

Общая трудоемкость дисциплины составляет 4 зачетных единиц 144 акад. часов, в том числе:

– контактная работа – 10,7 акад. часов:

– аудиторная – 10 акад. часов;

– внеаудиторная – 0,7 акад. часов

– самостоятельная работа – 129,4 акад. часов;

Форма аттестации - зачет с оценкой

Раздел/ тема дисциплины	Курс	Аудиторная контактная работа (в акад. часах)			Самостоятельная работа студента	Вид самостоятельной работы	Форма текущего контроля успеваемости и промежуточной аттестации	Код компетенции
		Лек.	лаб. зан.	практ. зан.				
1. Введение								
1.1 История микропроцессорной техники	4				10	Прочтение лекционного материала	Устный опрос (собеседование)	ПК-2.1
1.2 Современный этап развития микропроцессорных систем					10	Прочтение лекционного материала	Устный опрос (собеседование)	ПК-2.1
Итого по разделу					20			
2. Язык программирования С								
2.1 Основные термины и понятия	4	1	1		10	Подготовка к лабораторной работе	Защита лабораторной работы	ПК-2.1
2.2 Типы данных		1	2/2И		10	Подготовка к лабораторной работе	Защита лабораторной работы	ПК-2.1
2.3 Переменные, константы, массивы, указатели			2/2И		10	Подготовка к лабораторной работе	Защита лабораторной работы	ПК-2.1
2.4 Структуры					5	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
2.5 Циклы, ветвления					10	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
2.6 Функции					6	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
Итого по разделу		2	5/4И		51			
3. Среда разработки программного обеспечения Qt Designer								
3.1 Создание и настройка проекта	4				10	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
3.2 Отладка программы					5	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
Итого по разделу					15			
4. Микроконтроллер AVR Atmega8								

4.1 Описание, структура, характеристики	4				6	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
4.2 Работа с периферией контроллера					10	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
Итого по разделу					16			
5. Микроконтроллер ARM STM32F407								
5.1 Описание, структура, характеристики	4	2			8	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
5.2 Работа с периферией контроллера					11,2	Прочтение дополнительной литературы	Устный опрос (собеседование)	ПК-2.1
5.3 Среда проектирования STM32 CubeMX			1		8,2	Подготовка к лабораторной работе	Защита лабораторной работы	ПК-2.1
Итого по разделу		2	1		27,4			
6. Контроль								
6.1 Контроль	4							
Итого по разделу								
Итого за семестр		4	6/4И		129,4		зачет с оценкой	
Итого по дисциплине		4	6/4И		129,4		зачет с оценкой	

5 Образовательные технологии

Для реализации предусмотренных видов учебной работы в качестве образовательных технологий в преподавании дисциплины «Основы микропроцессорной техники» используются традиционная и модульно-компетентностная технологии.

Лекции проходят в традиционной форме и в форме лекций-консультаций. На лекциях-консультациях изложение нового материала сопровождается постановкой вопросов и дискуссией в поисках ответов на эти вопросы.

При выполнении лабораторных работ студенты учатся практическим навыками проектирования и моделирования устройств, рассмотренных на лекционных занятиях. При защите лабораторных работ перед студентами ставятся задачи, требующие логического мышления, принципа обобщения и сопоставления.

Самостоятельная работа стимулирует студентов в процессе подготовки домашних заданий, при решении задач на лабораторных занятиях, при подготовке к итоговой аттестации.

6 Учебно-методическое обеспечение самостоятельной работы обучающихся

Представлено в приложении 1.

7 Оценочные средства для проведения промежуточной аттестации

Представлены в приложении 2.

8 Учебно-методическое и информационное обеспечение дисциплины (модуля)

а) Основная литература:

1. Смирнов, Ю. А. Основы микроэлектроники и микропроцессорной техники : учебное пособие / Ю. А. Смирнов, С. В. Соколов, Е. В. Титов. — 2-е изд., испр. — Санкт-Петербург : Лань, 2013. — 496 с. — ISBN 978-5-8114-1379-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/12948 (дата обращения: 06.11.2020). — Режим доступа: для авториз. пользователей.

б) Дополнительная литература:

1. Смирнов, Ю. А. Технические средства автоматизации и управления : учебное пособие / Ю. А. Смирнов. — 3-е изд., стер. — Санкт-Петербург : Лань, 2020. — 456 с. — ISBN 978-5-8114-5413-6. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/140779 (дата обращения: 06.11.2020). — Режим доступа: для авториз. пользователей.

2. Лукинов, А. П. Проектирование мехатронных и робототехнических устройств : учебное пособие / А. П. Лукинов. — Санкт-Петербург : Лань, 2012. — 608 с. — ISBN 978-5-8114-1166-5. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/2765 (дата обращения: 10.11.2020). — Режим доступа: для авториз. пользователей.

в) Методические указания:

Приведены в приложении 3.

г) Программное обеспечение и Интернет-ресурсы:

Программное обеспечение

Наименование ПО	№ договора	Срок действия лицензии
MS Windows 7 Professional(для классов)	Д-1227-18 от 08.10.2018	11.10.2021
MS Office 2007 Professional	№ 135 от 17.09.2007	бессрочно
7Zip	свободно распространяемое ПО	бессрочно
Borland Turbo C++	№112301 от 23.11.2005	бессрочно
FAR Manager	свободно распространяемое ПО	бессрочно

Профессиональные базы данных и информационные справочные системы

Название курса	Ссылка
Электронная база периодических изданий East View Information Services, ООО «ИВИС»	https://dlib.eastview.com/
Национальная информационно-аналитическая система – Российский индекс научного цитирования (РИНЦ)	URL: https://elibrary.ru/project_risc.asp
Поисковая система Академия Google (Google Scholar)	URL: https://scholar.google.ru/
Информационная система - Единое окно доступа к информационным ресурсам	URL: http://window.edu.ru/

9 Материально-техническое обеспечение дисциплины (модуля)

Материально-техническое обеспечение дисциплины включает:

Тип и название аудитории	Оснащение аудитории
Учебные аудитории для проведения занятий лекционного типа	мультимедийные средства хранения, передачи и представления информации
Учебная аудитория для проведения лабораторных занятий: лаборатория автоматизированного электропривода постоянного и переменного тока	компьютеры Syntex mod-1+ LCD LG TFT19; лабораторный стенд №1; лабораторный стенд №2; стенд ШЭП-ПЧ «Исследование электроприводов постоянного тока»
Учебные аудитории для групповых и индивидуальных консультаций, текущего контроля и промежуточной аттестации	Доска, мультимедийный проектор, экран
Учебные аудитории для самостоятельной работы обучающихся	Персональные компьютеры с ПО из п. 8(г), выходом в Интернет и с доступом в электронную информационно-образовательную среду университета

ПРИЛОЖЕНИЕ 1

Учебно-методическое обеспечение самостоятельной работы обучающихся

По дисциплине «Основы микропроцессорной техники» предусмотрена аудиторная и внеаудиторная самостоятельная работа обучающихся.

Аудиторная самостоятельная работа студентов предполагает ответы на вопросы на лабораторных занятиях при защите работ.

Примерные вопросы для устного опроса и защиты лабораторных работ:

1. Чем отличается микроконтроллер от микропроцессора?

2. Назовите основные узлы и их назначение в структуре любого микропроцессора.

3. Что такое шина в микропроцессорной технике?

4. Назовите две основные архитектуры микропроцессоров. В чем их отличия?

5. Почему современные микропроцессоры содержат не одно ядро?

6. В чем заключается проблема дальнейшего роста тактовой частоты современных микропроцессоров?

7. Какие языки программирования в настоящее время используются для написания программ для микропроцессоров?

8. Что такое компилятор?

9. Опишите последовательность действий, выполняемых компилятором, при программировании микропроцессора.

10. Какие основные типы данных общеприняты при написании программ для микропроцессоров?

11. Опишите структуру проекта на языке C++. Что такое «заголовочный файл»?

12. Как и в каком месте программы объявляются переменные в языке C++?

13. Что происходит при объявлении переменных? Что такое инициализация переменной?

14. Что такое массив? Какие типы массивов вы знаете? Как задается массив?

15. Что такое указатель? Какие указатели бывают? Как они работают?

16. Что такое структуры в C++? Как объявить структуру?

17. Какие циклы языка C++ вы знаете? Приведите их синтаксис.

18. Какие ветвления в C++ вы знаете? Приведите их синтаксис.

19. Что такое функции в C++?

20. Опишите процесс создания и конфигурирования проекта в Qt Designer.

21. Чем отличается Qt Designer от других сред разработки (IDE)?

22. Чем характеризуется семейство микроконтроллеров AVR?

23. Объясните принцип работы с АЦП контроллера Atmega8.

24. Объясните принцип работы с таймером контроллера Atmega8.

25. Объясните принцип работы с портами ввода/вывода контроллера Atmega8.

26. Чем характеризуется семейство микроконтроллеров STM32F4?

27. Объясните принцип работы с АЦП контроллера STM32F407.

28. Объясните принцип работы с таймером контроллера STM32F407.

29. Объясните принцип работы с портами ввода/вывода контроллера STM32F407.

30. Какими периферийными устройствами обладает контроллер STM32F407?

ПРИЛОЖЕНИЕ 2

а) Планируемые результаты обучения и оценочные средства для проведения промежуточной аттестации:

Код индикатора	Индикатор достижения компетенции	Оценочные средства
ПК-2.1: Осуществляет подготовку технического задания на разработку системы электропривода		
ПК-2.1	Осуществляет подготовку технического задания на разработку системы электропривода	1. Чем отличается микроконтроллер от микропроцессора? 2. Назовите основные узлы и их назначение в структуре любого микропроцессора. 3. Что такое шина в микропроцессорной технике? 4. Назовите две основные архитектуры микропроцессоров. В чем их отличия? 5. Почему современные микропроцессоры содержат не одно ядро? 6. В чем заключается проблема дальнейшего роста тактовой частоты современных микропроцессоров? 7. Какие языки программирования в настоящее время используются для написания программ для микропроцессоров? 8. Что такое компилятор? 9. Опишите последовательность действий, выполняемых компилятором, при программировании микропроцессора. 10. Чем характеризуется семейство микроконтроллеров AVR? 11. Объясните принцип работы с АЦП контроллера Atmega8. 12. Объясните принцип работы с таймером контроллера Atmega8. 13. Объясните принцип работы с портами ввода/вывода контроллера Atmega8. 14. Какие основные типы данных общеприняты при написании программ для микропроцессоров? 15. Опишите структуру проекта на языке C++. Что такое «заголовочный файл»? 16. Как и в каком месте программы объявляются переменные в языке C++? 17. Что происходит при объявлении переменных? Что такое инициализация переменной? 18. Что такое массив? Какие типы массивов вы знаете? Как задается массив? 19. Что такое указатель? Какие указатели бывают? Как они работают? 20. Что такое структуры в C++? Как объявить структуру? 21. Какие циклы языка C++ вы знаете? Приведите их синтаксис. 22. Какие ветвления в C++ вы знаете? Приведите их синтаксис. 23. Что такое функции в C++? 24. Чем характеризуется семейство микроконтроллеров STM32F4? 25. Объясните принцип работы с АЦП контроллера STM32F407. 26. Объясните принцип работы с таймером контроллера STM32F407. 27. Объясните принцип работы с портами ввода/вывода контроллера STM32F407. 28. Какими периферийными устройствами обладает контроллер STM32F407? 29. Опишите процесс создания и конфигурирования проекта в Qt Designer. 30. Чем отличается Qt Designer от других сред разработки (IDE)? 31. Опишите процесс создания и конфигурирования проекта в STM32 CubeMX.

б) Порядок проведения промежуточной аттестации, показатели и критерии оценивания:

Промежуточная аттестация по дисциплине «Основы микропроцессорной техники» включает теоретические вопросы, позволяющие оценить уровень усвоения обучающимися знаний, и практические задания, выявляющие степень сформированности умений и владений.

Показатели и критерии аттестации (зачет с оценкой):

– на оценку **«отлично»** (5 баллов) – обучающийся демонстрирует высокий уровень

сформированности компетенций, всестороннее, систематическое и глубокое знание учебного материала, свободно выполняет практические задания, свободно оперирует знаниями, умениями, применяет их в ситуациях повышенной сложности.

– на оценку **«хорошо»** (4 балла) – обучающийся демонстрирует средний уровень сформированности компетенций: основные знания, умения освоены, но допускаются незначительные ошибки, неточности, затруднения при аналитических операциях, переносе знаний и умений на новые, нестандартные ситуации.

– на оценку **«удовлетворительно»** (3 балла) – обучающийся демонстрирует пороговый уровень сформированности компетенций: в ходе контрольных мероприятий допускаются ошибки, проявляется отсутствие отдельных знаний, умений, навыков, обучающийся испытывает значительные затруднения при оперировании знаниями и умениями при их переносе на новые ситуации.

– на оценку **«неудовлетворительно»** (2 балла) – обучающийся демонстрирует знания не более 20% теоретического материала, допускает существенные ошибки, не может показать интеллектуальные навыки решения простых задач.

– на оценку **«неудовлетворительно»** (1 балл) – обучающийся не может показать знания на уровне воспроизведения и объяснения информации, не может показать интеллектуальные навыки решения простых задач.

Обучающийся получает отметку **«зачтено»** при условии выполнения и защиты всех предусмотренных лабораторных работ на оценку не ниже «удовлетворительно». При зачете оценка ставится согласно среднему балу оценок защит лабораторных работ.

ПРИЛОЖЕНИЕ 3

3. ОСНОВЫ МИКРОПРОЦЕССОРНОЙ ТЕХНИКИ

ЛАБОРАТОРНАЯ РАБОТА №13

«Установка и настройка среды разработки *Qt Creator*»

Цель работы: произвести установку среды разработки программ *Qt Creator* и настроить комплект для разработки программ на языках программирования *C/C++*.

Введение

Qt Creator – это кроссплатформенная среда разработки программ, поддерживающая языки программирования *C, C++, Java, Python*.

Qt Creator позволяет разрабатывать программы для самых различных устройств: от микроконтроллеров до привычных операционных систем (далее ОС), таких как *Android*, *Windows* и *Linux*. При использовании графических элементов приложений ОС (кнопки, поля ввода текста и т.д.) используется библиотека *Qt*, содержащая необходимые для работы этих элементов функции.

На каждую ОС или контроллер необходим соответствующий набор компиляторов и отладчиков, составляющих так называемый *Комплект* (*Профиль*) *Qt*. Это позволяет написать программу один раз, а затем скомпилировать ее под ту или иную ОС, что и называется кроссплатформенностью.

Qt Creator имеет несколько бесплатных лицензий с открытым исходным кодом (*Open Source*).

Порядок выполнения работы

Для скачивания *Qt Creator* необходимо перейти по ссылке https://www.qt.io/offline-installers и выбрать версию в зависимости от ОС.

По окончании загрузки следует запустить загруженный файл и выполнить инструкции установщика. При установке потребуется зарегистрироваться в разделе «*Sign-up*», после чего указать путь для установки. Общее правило для всех сред разработки – в пути не должно содержаться русских символов.

На следующем экране нужно выбрать компоненты для установки согласно разрядности ОС, как показано на рис. 13.1.

Компонент *MinGW* – это компилятор языка программирования *C/C++* для ОС *Windows*.

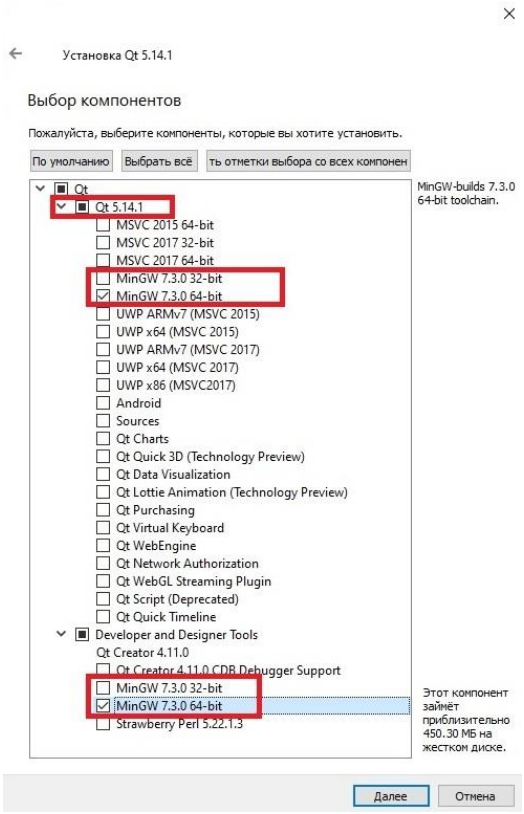

Рис. 13.1. Выбор компонентов для установки

После установки следует запустить *Qt Creator* и перейти в меню *Инструменты->Параметры->Комплекты*, чтобы убедиться в корректной автоматической настройке комплекта, как показано на рис. 13.2.

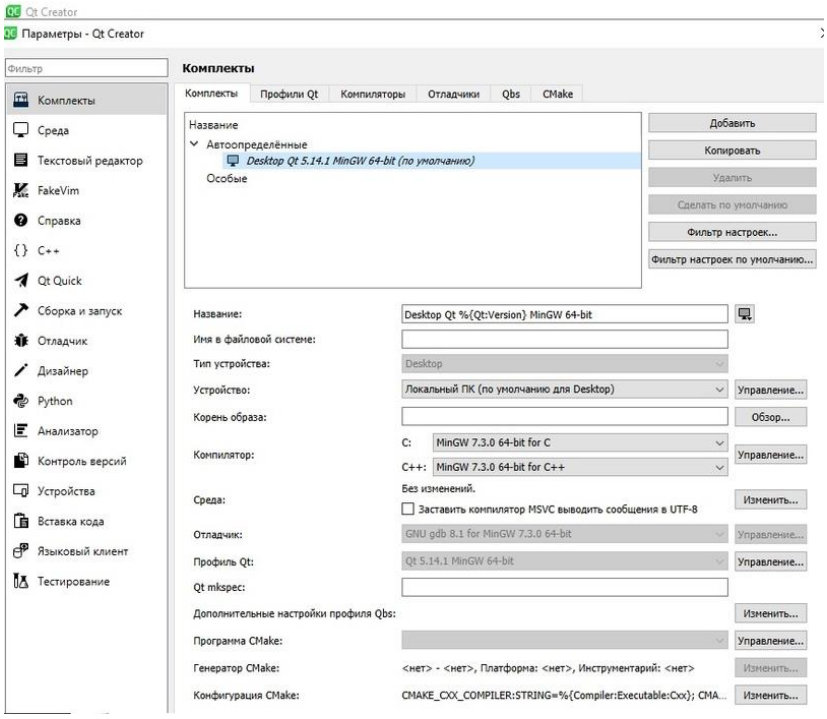

Рис. 13.2. Окно настройки комплектов

Для написания программ контроллеров семейства *AVR* в среде *Qt Creator* необходимо создать и настроить так называемый комплект, включающий в себя компиляторы avr-gcc.exe для языка *C* и *avr-g++.exe* для *C++*, а также отладчик *avr-gdb.exe*.

Вторым важным этапом является создание и сборка проекта. Встроенные средства даже последних версий *Qt Creator* не позволяют это сделать стандартными методами через меню *Файл-Новый...* Для подобных целей в *Qt Creator* предусмотрен другой механизм.

В самом начале изучения языка программирования Си говорилось, что написать исходный файл программы можно даже в самом простом текстовом приложении. Но запустить и исполнить файл с исходным текстом нельзя, для этого его нужно скомпилировать.

Сделать это можно из командной строки (для ОС *Windows* запуск: *Пуск-Выполнить cmd.exe*), написав в ней:

```
gcc имя_исходного_файла.c -o имя_исполняемого_файла
```

Указанная строка говорит операционной системе, что нужно:

- запустить компилятор *gcc* языка Си;

- из файла *имя_исходного_файла.c* сделать исполняемый файл с именем *имя_исполняемого_файла*.

Нами также был рассмотрен механизм языка Си, использующий разбиение проекта на разные файлы с исходными текстами с расширением .c и заголовочные файлы с расширением *.h*.

Если бы программа содержала несколько файлов с исходными текстами и заголовочные файлы, содержащиеся, например, в папке проекта с именем *include*, то скомпилировать ее с помощью командной строки можно было бы так:

```
gcc имя_исходного_файла1.c имя_исходного_файла2.c -o имя_исполняемого_файла -I./include
```

Скомпилировать сложный проект, содержащий более десятка разных файлов, из командной строки очень не просто, особенно если он использует различные стандартные библиотеки Си.

Любая среда разработки выполняет все эти операции автоматически, когда мы нажимаем кнопку «*Собрать»* (*Build*). Все пути к стандартным библиотекам языка обычно настраиваются по умолчанию, а компиляторы указываются подобно комплектам в *Qt Creator*.

Но если программа создается для какого-либо микроконтроллера, то к стандартным библиотекам обязательно нужно добавлять исходные и заголовочные файлы именно для этого контроллера. В таких файлах содержится полный инструментарий для работы с регистрами, портами и периферией контроллера. И именно этого по умолчанию нет в составе большинства универсальных сред, таких как *Qt Creator*.

Тот самый «другой механизм», о котором говорилось в самом начале, позволяет сообщить *Qt Creator* все настройки проекта: модель контроллера, компиляторы, пути к библиотекам и файлам с исходными кодами, заголовочным файлам проекта и библиотек контроллера. Называется такой механизм *Makefile*, он выполняется специальной программой *make*.

По умолчанию в ОС *Windows* отсутствует такая программа, т.к. все программы в этой ОС уже скомпилированы, их исходный код в большинстве случаев является коммерческим. Надобности для большинства пользователей *Windows* в системе сборки *make* нет.

В отличие от *Windows* в ОС *Linux* все программы, как и сам *Linux*, имеют открытый исходный код, а система сборки входит в большинство дистрибьютивов по умолчанию.

При установке Qt Creator была также установлена система сборки и компиляции для *Windows MinGW*. В директории *MinGW* содержится файл *MinGW32-make.exe*, который является нужной нам системой сборки для контроллеров *AVR*.

Скопируем этот файл в любое место и переименуем его в *make.exe*. Поместим его обратно в ту же папку *MinGW*. Добавим путь к этой папке в «Переменные окружения *Windows*». Перезагрузим компьютер. Эти действия необходимы, чтобы *Qt Creator* мог найти утилиту *make.exe*.

После этого необходимо получить у преподавателя архив *atmega16 (Win).zip*, разархивировать его в папку с проектами *Qt Creator*, запустить *Qt Creator* и открыть проект, выбрав файл *atmega16.creator*.

В архиве содержится шаблон проекта для контроллера *ATmega16*. Основные настройки находятся в файле *Makefile*. В нем следует изменить пути к файлам и папкам в зависимости от их расположения на компьютере.

Контрольные вопросы

1. Что такое *Qt Creator*? Для чего он используется?
2. Почему *Qt Creator* называют кроссплатформенным?
3. Что необходимо сделать, чтобы написанная в *Qt Creator* программа была совместима с разными ОС?

ЛАБОРАТОРНАЯ РАБОТА №14

«Введение в язык программирования *C*. Простейшая программа»

Цель работы: изучение основных терминов и понятий языка программирования *C*, приобретение навыков написания программ на языке *C*.

Введение

Язык программирования *C* (Си) был разработан сотрудником корпорации *Bell Labs* (прежнее название *AT&T Bell Laboratories*) Дэннисом Ритчи в 1973 году для написания операционной системы *Unix*. Позже *C* был адаптирован под множество других платформ и операционных систем, получил в своем развитии несколько редакций и стал основой для языков *C#, C++, Java*.

Языки программирования можно разделить на две группы: компилируемые и интерпретируемые. В интерпретируемых языках (*PHP, Perl, Python*) код программы выполняется с помощью специальной программы-интерпретатора построчно. В компилируемых языках (*C, C++*) весь исходный код программы переводится в машинные коды с помощью компилятора. Программы, написанные на таких языках, выполняются значительно быстрее.

Программы можно писать в любом текстовом редакторе, наиболее удобными из которых являются *Sublime Text 3, Notepad++*, или использовать для этого среду разработки (*IDE – Integrated Development Environment*), например *Qt Creator, NetBeans, Visual Studio Code*. Для компиляции исходных файлов с программами и получения запускаемого файла необходим компилятор. Самым популярным компилятором является *GCC*, входящий в бесплатный пакет средств разработки программ *MinGW* (для ОС *Windows*).

Для написания программ на *C* могут быть использованы все буквы английского алфавита (заглавные и строчные), цифры и специальные символы (, . + - * ^ & = ~ < > ( ) { } [ ] | % ! ? ‘ “ : ; _ / \ #).

Программы строятся из подпрограмм, которые принято называть функциями. Язык *C* предполагает следующую структуру функций:

```
тип имя (аргументы)
{
    инструкции, операции, вызовы функций и т.д.;
```

```
    return значение;
}
```

В приведенной структуре *тип* – наименование набора возможных значений, возвращаемых функцией в результате ее выполнения, *аргументы* – передаваемые в функцию переменные, необходимые для ее работы, *return* – оператор, осуществляющий выход из функции. Аргументы функции и значение после *return* не являются обязательными. Следует отметить, что некоторые компиляторы выдают предупреждение или даже ошибку при отсутствии аргументов функции. В этом случае вместо аргументов указывают ключевое слово *void* (от англ. «пустой»). После return значения может не быть в том случае, если функция не возвращает значение результата, при этом тип, указанный перед именем функции должен быть указан словом *void*. Тело функции, как и любого блока инструкций языка С, заключается в фигурные скобки { }.

При выполнении программы первой выполняется главная функция с именем *main*.

```
int main()
{
    //Код программы
    return 0;
}
```

При написании программы под операционную систему *Windows* или *Linux* функция *main* должна иметь тип возвращаемого значения *int* (целочисленный тип данных), т.е. после выполнения программа должна вернуть операционной системе результат корректного завершения, в противном случае ОС фиксирует ошибку приложения. В программах, написанных для микроконтроллеров, часто главная функция имеет заголовок void main(void) из-за отсутствия ОС как таковой.

В приведенном примере главной функции указан комментарий вида *«//Код программы»*. Две косые черты обозначают комментарий до конца строки, т.е. все, что находится после // до конца строки компилятором будет проигнорировано. Знаками /* и */ обозначают начало и окончание блока комментария с произвольным количеством строк.

Каждая инструкция программы должна заканчиваться точкой с запятой «;».

Файл, содержащий исходный код программы, должен иметь расширение «*.c*». Кроме файлов с исходными кодами в языке *C* используют заголовочные файлы, имеющие расширение «*.h*». В них содержатся только заголовки (описания) функций, которые могут быть использованы в программе. Сами же функции содержатся в одноименном заголовочному файлу с расширением «*.c*». Для подключения заголовочного файла применяют директиву *#include <название файла.h>* в самом начале файла с исходным кодом программы.

Подобный механизм очень удобен в больших по объему кода проектах, выглядит он, как показано на рис. 14.1.

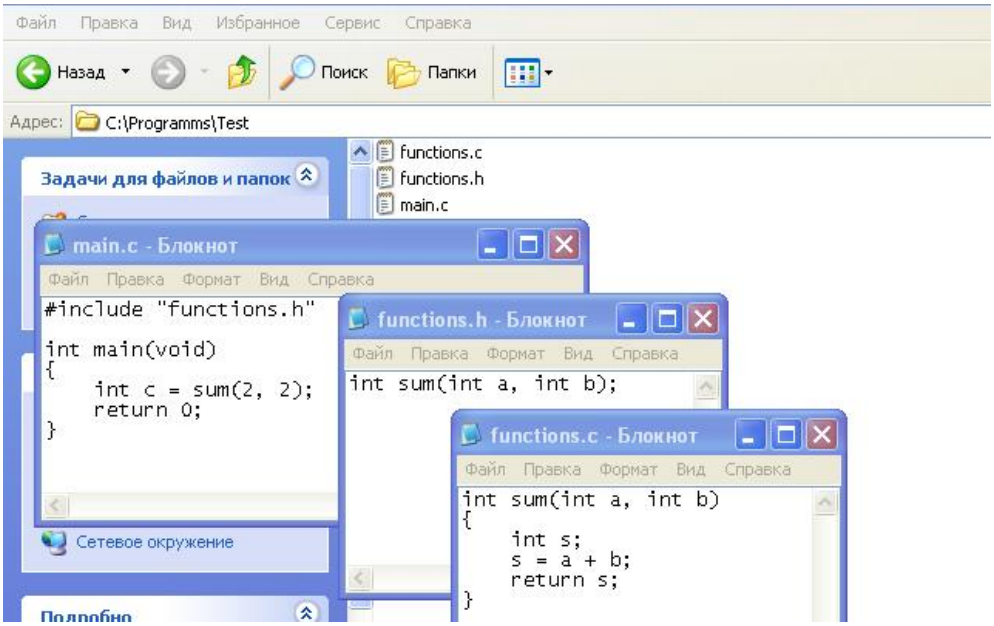

Рис. 14.1. Структура файлов в языке *C*

Директива *#include* сообщает специальной программе, обрабатывающей текст исходного кода и называющейся препроцессором, что в коде содержатся функции, описание которых находится в файле *functions.c*. Препроцессор выполняет обработку исходного кода до компилятора. Каждая директива #include размещается с новой строки, поэтому в конце строки с директивой #include знак «;» не ставится.

Отметим еще одну особенность директивы *#include*, состоящую в том, что имя указанного за ней заголовочного файла может быть заключено в кавычки или в угловые скобки. В первом случае препроцессор будет искать указанный заголовочный файл в локальной папке проекта, а во втором – в папке среды разработки.

При выполнении программы на рис. 14.1 переменной *c* будет присвоен результат выполнения функции *sum*, аргументами которой будут два числа 2 и 2.

Существует большое количество рекомендаций по стилю написания кода программ. Эти рекомендации позволяют сделать код максимально удобным для восприятия, повысить его читаемость. Следует ознакомиться с подобными рекомендациями самостоятельно.

Порядок выполнения работы

1. Создать новую папку для проекта. Рекомендуется не использовать в пути русские буквы.
2. Создать в папке текстовый файл, изменить его имя по Вашему усмотрению и расширение «*.c*».
3. Написать в файле следующий листинг:

```
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

4. Сохранить изменения в файле. Запустить консоль (*Пуск->Выполнить...->cmd.exe*).
5. Перейти в папку проекта (*cd [имя диска]:\[имя папки]*).
6. Выполнить команду: *gcc [имя файла.c]*. Программа будет скомпилированна, если не возникло ошибок, и в папке с программой автоматически будет создан файл *a.exe*.

7. Запустить в консоли вновь созданный файл: *a [нажать Enter]*. В консоли будет выведено *Hello World*!

8. Привести в отчете по работе листинг программы, дополненный комментариями.

Контрольные вопросы

1. Что такое компилятор? Чем отличаются компилируемые языки программирования от интерпретируемых?

2. Что такое функция в языке *C*? В чем отличие главной функции? Объясните структуру функции.

3. Что такое заголовочный файл? Каково его назначение?

4. Что такое директива *#include*? Для чего она используется? Какие способы указания заголовочных файлов Вы знаете? В чем их отличия?

5. Какие способы написания комментариев на языке *C* Вы знаете?

6. Что такое оператор *return*? Объясните его использование.

7. Какой компилятор является наиболее распространенным в ОС *Windows*? Как создать исполняемый файл с помощью консоли ОС *Windows*? Какое имя по умолчанию получает исполняемый файл после компиляции?

ЛАБОРАТОРНАЯ РАБОТА №15

«Изучение типов данных языка *C*»

Цель работы: изучение основных типов данных, определение понятия переменной и размещения переменных в памяти.

Введение

Все данные, с которыми работает программа, хранятся в ячейках оперативной памяти. Каждая ячейка памяти имеет свой адрес, подобно ячейкам таблицы, находящимся на пересечении строк и столбцов. Если программа работает под операционной системой (*Windows, Linux*), то при каждом запуске этой программы ее данные будут размещаться в ячейках памяти с разными адресами, свободными в момент запуска. После завершения программы ячейки памяти освобождаются и могут использоваться ОС по необходимости.

Данные представляют собой двоичные коды, количество бит которых кратно восьми, т.е. одному байту. Данные могут иметь разную длину: 1 байт, 2 байта, 4 байта.

Переменными называют данные, значения которых могут меняться в ходе работы программы. *Константы* – это данные, значение которых постоянно, определяется в начале выполнения программы и не может быть изменено при ее дальнейшем выполнении.

Обращаться к данным по адресу, который при каждом запуске программы меняется, невозможно. Поэтому данным принято давать имя, а чтобы определить их длину и допустимые операции над ними – тип.

Язык *C* часто называют языком со строгой статической типизацией. Это значит, что все переменные должны иметь определенный тип до компиляции, т.е. не может быть данных, тип которых будет определен при выполнении программы.

Имя переменной (идентификатор) может содержать буквы английского алфавита, цифры, знак подчеркивания. Причем одна и та же буква, записанная в разных регистрах (прописная или строчная), будет восприниматься как разные идентификаторы. Например, переменные «A» и «a» – две разные переменные. Имя не может начинаться с цифры. Принято записывать имена переменных по-английски в смешанном регистре, начиная с нижнего (правильно – *book, bookNumber*, неправильно – *Kniga, nomerKnigi*).

В языке *C* имеется набор зарезервированных слов, которые не могут являться названием переменной: *auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while*.

Базовые типы данных языка *C* следующие:

- ***char*** – целые числа от -128 до 127 (8 бит);
- ***unsigned char*** – целые числа от 0 до 255 (8 бит);
- ***short*** – целые числа от -32 768 до 32 767 (16 бит);
- ***unsigned short*** – целые числа от 0 до 65 535 (16 бит);
- ***int*** – целые числа от −2 147 483 648 до 2 147 483 647 (32 бита) для 32- и более -разрядных систем, и от -32 768 до 32 767 (16 бит) для 8- и 16-битных систем;
- ***unsigned int*** – аналогично int от 0 до 4 294 967 295 или до 65 535;
- ***long int*** – целые числа от −2 147 483 648 до 2 147 483 647 (32 бита);
- ***unsigned long int*** – целые числа от 0 до 4 294 967 295 (32 бита);
- ***long long int*** – целые числа от −9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 (64 бита);
- ***unsigned long long int*** – целые числа от 0 до 18 446 744 073 709 551 615 (64 бита);
- ***float*** – вещественное число с плавающей точкой в диапазоне +/- 3.4E-38 до 3.4E+38 (32 бита);
- ***double*** – вещественное число с плавающей точкой в диапазоне +/- 1.7E-308 до 1.7E+308 (64 бита);
- переменная символьного типа хранит код символа (буквы, цифры, знака и т.д.), занимает в памяти 8 бит, представляется типом ***char***.

Как видно, некоторые типы данных занимают разное количество памяти в зависимости от платформы. Для случаев, когда необходимо точно знать размер типа в байтах, в языке *C* предусмотрен оператор *sizeof(имя переменной)*, возвращающий размер типа указанной в скобках переменной.

Прежде чем использовать переменную в программе, ее необходимо объявить, т.е. указать ее тип и имя.

```
char a;
int b = 10;
```

Инициализацией переменной называют присвоение ей значения. Это можно сделать при объявлении, как с переменной *b* в примере выше, или позже в коде программы. Неинициализированная переменная хранит «мусор», оставшийся в ячейках оперативной памяти.

Следует выбирать тип переменной с учетом ее возможных значений. Например, переменная, хранящая количество студентов группы, не может принимать отрицательные значения и вряд ли превысит число 255. Поэтому нецелесообразно расходовать оперативную память, обозначая такую переменную типом *int*. Для этого лучше использовать тип *unsigned char*.

Если инициализировать переменную значением вне диапазона типа (unsigned char a = 256;), то на этапе компиляции программы возникнет ошибка. Однако переполнение типа переменной может произойти при выполнении программы. Например,

```
unsigned char a = 255, b = 1, c;
c = a + b;
```

В программе объявлены 8-битные переменные «a, b, c», с возможным диапазоном значений 0..255. После выполнения программы в переменной «c» не будет значение 256, как это ожидается, а будет храниться 0. Сложим в двоичной системе числа 255 и 1.

```
    1111 1111 (255)
+
    0000 0001 (1)
```

10000 0000 (256)

Единица самого старшего разряда не помещается в 8 бит переменной «c», поэтому в ней останутся только нули. Следует учитывать этот факт, чтобы избежать возможных коллизий программы.

Как упоминалось ранее, переменная типа *char* может быть использована для хранения кода символа. Инициализировать такую переменную можно так:

```
char c = 'f';
```

В переменной «c» будет храниться число 102, соответствующее символу «f» таблицы ascii (англ. *American standard code for information interchange*).

В языке С отсутствует логический тип данных. Вместо него могут быть использованы целочисленные типы. Значению «ложь» соответствует ноль, «истине» – любое целое число, отличное от нуля. Однако для удобства написания программ и наглядности в языке *C* присутствует стандартный заголовочный файл *stdbool.h*, содержащий макросы, определяющие тип *bool*, значение *true* (истина) и *false* (ложь). Для его подключения следует использовать директиву *#include*.

Операции и их обозначения в *C* следующие:

+ – сложение;

- – вычитание;

***** – умножение;

/ – деление;

% – остаток от деления;

++ – инкрементирование (увеличение на 1):

```
i++; // прибавить к значению переменной i единицу
++i; // прибавить к значению переменной i единицу
```

-- – декрементрирование (уменьшение на 1):

```
i--; // уменьшить значение переменной i на единицу
--i; // уменьшить значение переменной i на единицу
```

= – операция присвоения значения;

== – сравнение на равенство, возвращает 1, если сравниваемые значения (*операнды*) равны, 0 – в противном случае;

> – возвращает 1, если первый операнд больше второго;

< – возвращает 1, если первый операнд меньше второго;

>= – возвращает 1, если первый операнд больше или равен второму;

<= – возвращает 1, если первый операнд меньше или равен второму;

!= – возвращает 1, если первый операнд не равен второму;

! – отрицание;

&& – возвращает 1, если оба условия равны истине:

```
if ( (a == b) && (b > 0))...; //если a равно b и b больше 0
```

& – конъюнкция, логическое умножение;

|| – возвращает 1, если хотя бы одно условие равно истине:

```
if ( (a == b) || (b > 0)) ...; //если a равно b или b больше 0
```

| – дизъюнкция, логическое сложение;
^ – исключающее или;
~ – поразрядная инверсия (отрицание);
<< – побитовый сдвиг влево на заданное количество разрядов:

```
int a = 3 << 2;          // 3₁₀ (в двоичной системе 0011₂)
                         // сдвинуть влево на 2 разряда
                         // результат 1100₂ – 12₁₀
```

>> – побитовый сдвиг вправо на заданное количество разрядов.
Арифметические, логические и сдвиговые операции можно объединить с присваиванием:

```
a += 2;          // равносильно a = a + 2
b &= 3;          // равносильно b = b & 3
c <<= 1; // равносильно c << 1
```

Важной особенностью языка *C* является понятие *области видимости переменных*. Этим термином определяют блоки программы, в которых объявленные переменные доступны. Поясним это примером с комментариями:

```
int main(void)
{
    unsigned char a;                //объявили переменную a, она
                                    //доступна во всех вложенных блоках
                                    //функции main
    for (a = 0; a < 10; a++)        //цикл на 10 повторений является
                                    //вложенным по отношению к main
    {
        unsigned char b;   //объявили переменную b
        b = a;                      //переписали значение из a в b
    }                               //конец цикла
    b = 0;                          //ошибка – переменная b доступна
                                    //только в том блоке, где она была
                                    //объявлена, т.е. в цикле for
    return 0;
}
```

Таким образом, переменные, объявленные в каком-либо вложенном блоке, доступны только внутри этого блока или во вложенных в него блоках, но не за границами этого блока.

Возможно объявление и инициализация переменной даже до главной функции main. Такие переменные доступны во всех функциях и блоках файла программы и называются *глобальными переменными*. Следует избегать использования глобальных переменных, т.к. очень сложно отследить все изменения такой переменной при большом количестве функций программы, сложно найти ошибку в коде.

Для задания именованных констант в языке *C* используется директива препроцессора *#define*:

```
#define ИМЯ_КОНСТАНТЫ значение
```

В сообществе разработчиков принято записывать имя константы в верхнем регистре с нижним подчеркиванием в качестве разделителя.

Область видимости константы аналогична области видимости переменной.

Отменить константу можно с помощью директивы *#undef*:

```
#undef ИМЯ_КОНСТАНТЫ
```

Порядок выполнения работы

1. Создайте новый файл для программы.
2. Внесите в него следующий листинг программы:

```
int main()
{
    unsigned char a = 15
    int b = 138;
    if (a<b);
    a = b;
}
```

3. Скомпилируйте программу. Запишите вывод компилятора об ошибках.
4. Выясните смысл всех ошибок программы, используя поисковую систему (*Google*, *Yandex*).
5. Исправьте ошибки и добейтесь успешной компиляции программы.

Контрольные вопросы

1. Что такое переменная? Что значит объявить и инициализировать переменную?
2. Что такое область видимости переменной?
3. Что будет храниться в переменной, если ее объявить, но не инициализировать?
4. Что такое переполнение типа? Приведите пример результата переполнения.
5. Какие целочисленные типы данных Вы знаете? Сколько места в оперативной памяти они занимают?
6. Для чего используется модификатор *unsigned*?
7. Какие вещественные типы языка *C* Вы знаете?
8. С помощью какого оператора языка *C* можно точно определить размер типа переменной?
9. Почему не следует использовать глобальные переменные?
10. Как задать именованную константу? Как ее отменить?

ЛАБОРАТОРНАЯ РАБОТА №16

«Изучение массивов»

Цель работы: ознакомиться с понятием «массив», его структурой, способами объявления и инициализации, приобрести практические навыки работы с указателями языка *C*.

Введение

При написании программ часто приходится работать с большим количеством однотипных данных. Использование для подобных задач обыкновенных переменных приведет к увеличению листинга программы, частому повторению одних и тех же действий, но для разных переменных. В таких случаях значительно удобнее использовать массивы.

Массив - это набор объектов одного типа, расположенных в оперативной памяти непрерывно, к которым можно обращаться по общему имени.

Элементом массива называют объект, расположенный в памяти в границах массива. У каждого элемента массива есть свой *адрес*, совпадающий с адресом начальной ячейки памяти расположения этого элемента, *индекс*, являющийся порядковым номером элемента в массиве, и значение.

Имя массива - общий для всех элементов массива идентификатор.
Размер массива совпадает с количеством элементов массива.
Тип массива - тип элементов массива.
Длина массива - количество байт, занимаемое одним элементом массива, умноженное на количество элементов.
Массив может быть объявлен следующим образом:

тип имя[размер] = {инициализация элементов};

Пример:

```
char a[4] = {3, 12, 6, 9};
```

В примере объявлен массив с именем *a*, содержащий 4 элемента типа *char* (8 бит), значения элементов: *a*[0] = 3, *a*[1] = 12, *a*[2] = 6, *a*[3] = 9. Нумерация элементов массива начинается с нуля.
Если все элементы массива инициализированы в момент объявления массива, как в примере выше, то размер массива может не указываться, т.е. массив можно было бы объявить так:

```
char a[] = {3, 12, 6, 9};
```

Если при инициализации указаны значения не всех элементов, то их значения будут автоматически приравнены к нулю.
Инициализация элементов массива при объявлении необязательна, т.е. допустимо объявить массив, указав только его тип и имя с размером:

```
char a[4];
```

Массивы могут быть многомерными, тогда их объявление выглядит так:

тип имя[размер1][размер2]…[размерX];

При работе с массивами удобно использовать циклы, которые будут рассмотрены в следующих разделах дисциплины.

Порядок выполнения работы

1. Создать одномерный массив.
2. Организовать ввод элементов массива с клавиатуры функцией *scanf*().
3. Написать функцию нахождения элемента массива с минимальным значением.
4. В отчете представить листинг программы с комментариями.

Контрольные вопросы

1. Что такое массив?
2. С какого числа начинается нумерация элементов массива?
3. Какие способы объявления массива Вы знаете?
4. Обязательно ли указывать размер массива при его объявлении?
5. Что такое инициализация массива?

ЛАБОРАТОРНАЯ РАБОТА №17

«Изучение структур языка *C*»

Цель работы: усвоить назначение объекта «структура», изучить способы объявления структур и инициализации их полей.

Введение

Структура - это тип данных, предназначенный для объединения разных объектов под одним именем, которое является *типом структуры*. В качестве объектов могут выступать переменные, массивы, указатели и другие структуры.

Чтобы понять назначение структуры, рассмотрим пример и сравним реализацию программы с использованием простых переменных, массивов и структур.

Пусть требуется написать программу для хранения марки, модели, цвета и года выпуска автомобиля, введённых пользователем с клавиатуры.

Реализуем программу, использующую простые переменные, листинг которой показан на рис. 17.1.

```c
#include <stdio.h>

int main(void)
{
    char manufacturer[20];  //массив элементов для названия производителя длиной до 20 символов
    char model[20];         //массив элементов для названия модели длиной до 20 символов
    char color[10];         //массив элементов для названия цвета длиной до 10 символов
    unsigned int year;      //переменная для года выпуска
    printf("Enter manufacturer of the car: ");
    scanf("%s", manufacturer);              //ввод производителя
    printf("Enter model of the car: ");
    scanf("%s", model);                     //ввод модели
    printf("Enter color of the car: ");
    scanf("%s", color);                     //ввод цвета
    printf("Enter year of manufacturing the car: ");
    scanf("%d", &year);                     //ввод года выпуска
    printf("%s %s %s %d\n", manufacturer, model, color, year); //вывод всей информации на экран
    return 0;
}
```

```
Terminal
File  Edit  View  Search  Terminal  Help
Enter manufacturer of the car: nissan
Enter model of the car: cefiro
Enter color of the car: blue
Enter year of manufacturing the car: 1992
nissan cefiro blue 1992
Press <RETURN> to close this window...
```

Рис. 17.1. Листинг программы с простыми переменными

В приведенной программе возможна работа всего с одним абстрактным объектом (автомобилем), т.к. каждый его параметр хранится в отдельной, не связанной с другими, переменной.

Если таких объектов несколько, можно объявить массивы для хранения однотипной информации, как показано на следующем рисунке. Однако и в этом случае массивы не связаны между собой, и, при допущении логической ошибки в программе, возможна ошибочная работа с элементами разных массивов.

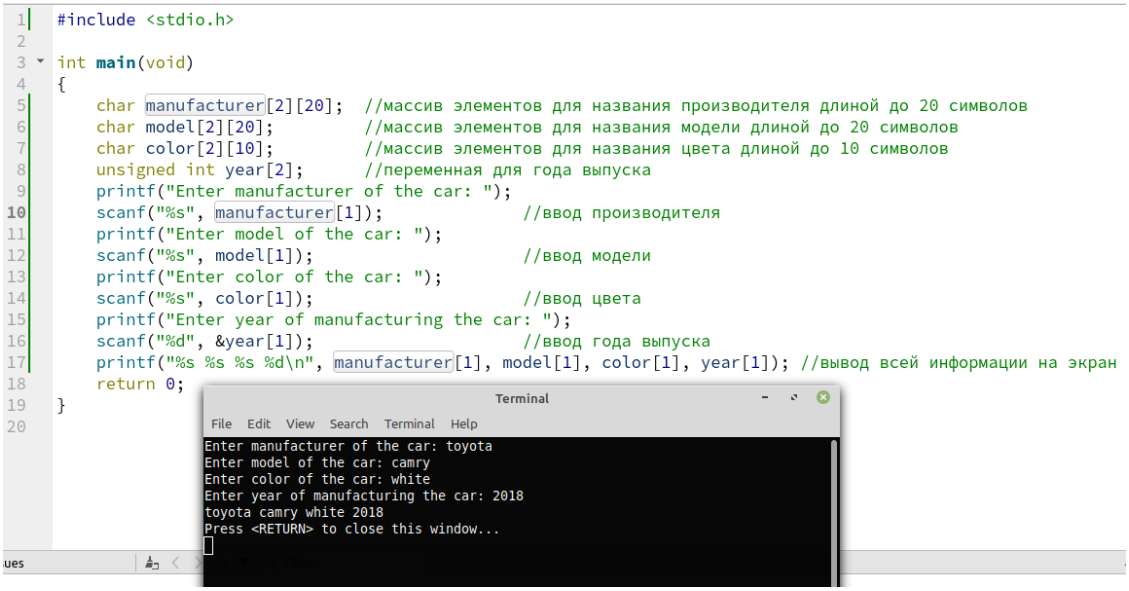

Рис. 17.2. Листинг программы с набором массивов

Структуры позволяют трактовать группу связанных между собой объектов не как множество отдельных элементов, а как единое целое. Структура представляет собой сложный тип данных, составленный из простых типов.

Структура объявляется следующим образом:

struct тип_структуры {

тип имяЭлемента1;

тип имяЭлемента2;

. . .

тип имяЭлементаN;

};

Напишем программу для рассмотренного ранее примера, но с использованием структуры.

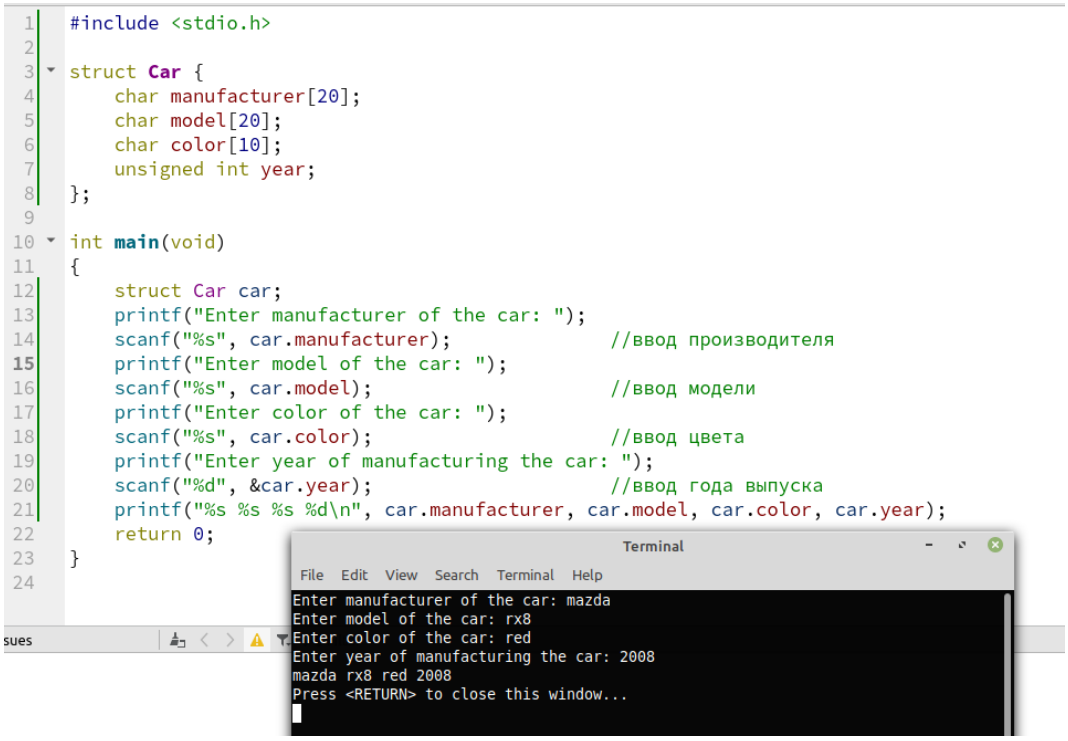

Рис. 17.3. Листинг программы с использованием структуры

В приведенной программе объявлена структура типа *Car*. Эта структура состоит из четырех элементов (полей структуры): производитель, модель, цвет и год выпуска.

В функции *main* объявлена переменная *car*, тип которой называется *Car*. Все поля структуры этой переменной относятся только к ней. Также имеется возможность работать с массивами структур.

```c
#include <stdio.h>

struct Car {
    char manufacturer[20];
    char model[20];
    char color[10];
    unsigned int year;
};

int main(void)
{
    struct Car car[4];
    int i;
    for (i = 0; i < 4; i++)
    {
        printf("Enter manufacturer of the car: ");
        scanf("%s", car[i].manufacturer);              //ввод производителя
        printf("Enter model of the car: ");
        scanf("%s", car[i].model);                     //ввод модели
        printf("Enter color of the car: ");
        scanf("%s", car[i].color);                     //ввод цвета
        printf("Enter year of manufacturing the car: ");
        scanf("%d", &car[i].year);                     //ввод года выпуска
    }
    printf("%s %s %s %d\n", car[1].manufacturer, car[1].model, car[1].color, car[1].year);
    return 0;
}

```

Рис. 17.4. Листинг программы с использованием массива структур

Обращение к элементу структуры осуществляется через точку после имени структуры. Имя структурной переменной может быть указано при объявлении структуры. В этом случае оно размещается после закрывающей фигурной скобки }.

При объявлении структур, их разрешается вкладывать одну в другую.

Доступ к элементам структуры можно осуществить с помощью указателей. Для этого необходимо инициализировать указатель адресом структуры. При этом обращение к полям структуры осуществляется через стрелку «->».

Порядок выполнения работы

1. Написать программу учета книг, используя структуры языка *C*.
2. Изменить программу таким образом, чтобы заполнение полей структуры осуществлялось вводом с клавиатуры.
3. Указать комментарии к программе.

Контрольные вопросы

1. Что такое «структура» в языке Си?
2. В чем отличие структуры от переменной или массива?
3. В каких случаях предпочтительно использование структур?

ЛАБОРАТОРНАЯ РАБОТА №18

«Язык программирования Си. Условные операторы»

Цель работы: усвоить синтаксис и особенности использования условных операторов языка Си.

Введение

При написании программы часто приходится сравнивать значения переменных и выполнять одни или другие действия в зависимости от результата сравнения. В таком случае говорят, что программа разветвляется. Для этих целей используют условные операторы.

Условный оператор if

Полная форма оператора выглядит следующим образом:

```
if (Условие)
{
    БлокОпераций1;
}
else
{
    БлокОпераций2;
}
```

Если *Условие* истинно, то выполнится *БлокОпераций1*, а иначе (т.е. *Условие* ложно) – *БлокОпераций2*. Блок операций может состоять из одной операции, тогда наличие фигурных скобок необязательно.

```
if (Условие)
    Операция1;
else
```

Операция2;

Если не требуется выполнять действия в случае ложного значения *Условия*, то блок *else* может быть исключен. Неполная форма оператора выглядит так:

```
if (Условие)
{
    БлокОпераций1;
}
```

В примере ниже объявляется переменная a, ей присвоено значение 7. Затем проверяется условие: если *a* больше 0, то вывести на экран значение переменной, иначе – значение переменной + 10. Результат выполнения программы – число 7.

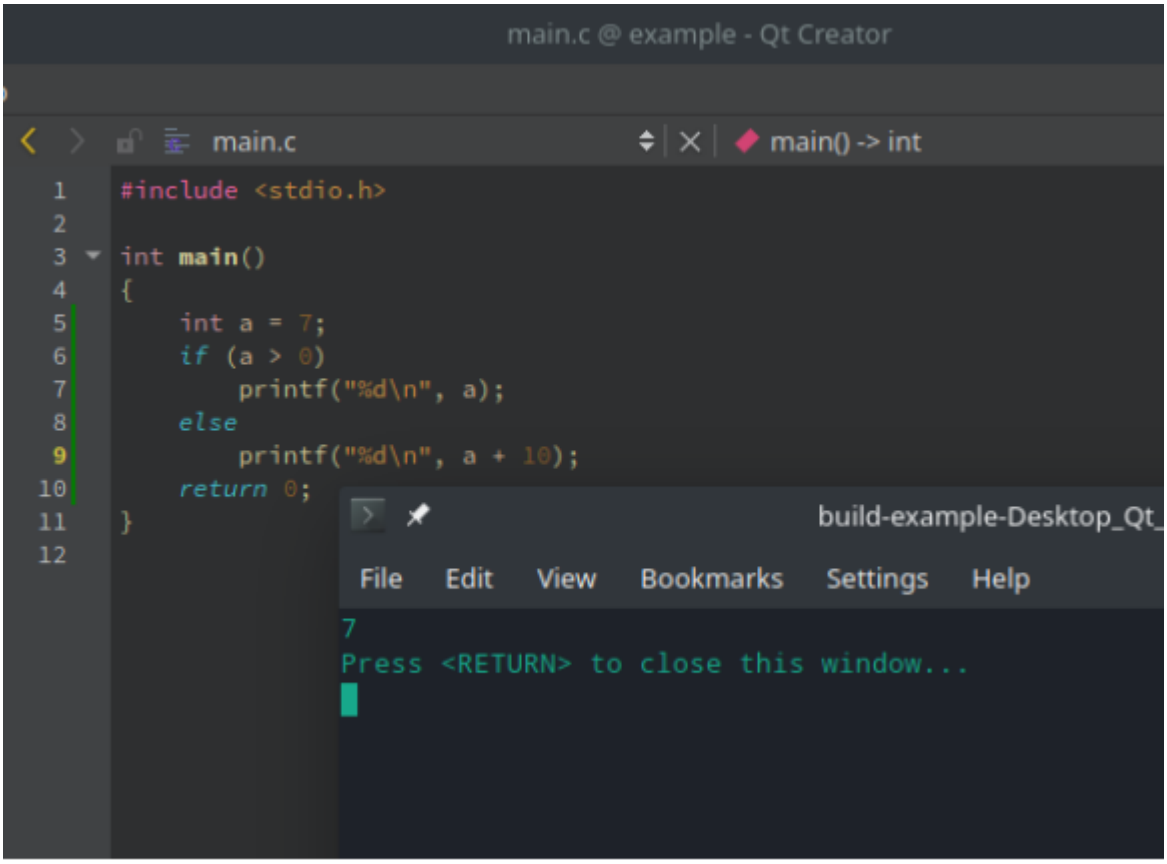

Рис.18.1. Пример условного оператора *if*

Оператор *if* может быть вложенным, т.е. после выполнения проверки одного условия проверяется следующее.

```
if (Условие1)
    if (Условие2)
        Операция1;
    else
        Операция2;
```

Обратим внимание, что в таком случае опция else относится к последнему *if*.

Оператор ветвления switch

В некоторых случаях требуется производить сравнение не с одним значением, а с некоторым их количеством. Использование для этого оператора *if* может оказаться не удобным, с точки зрения написания программы. Выбрать один из нескольких вариантов выполнения программы можно с помощью оператора *switch..case*.

```
switch (ЦелоеВыражение)
{
    case Значение1: БлокОпераций1;
    break;
    case Значение2: БлокОпераций2;
    break;
    . . .
    case Значениеn: БлокОперацийn;
    break;
    default: БлокОперацийПоУмолчанию;
    break;
}
```

Данный оператор работает следующим образом:
1. Вычисляется *ЦелоеВыражение*;
2. Полученное значение сравнивается со *Значениями*;
3. Если *Значение* найдено, то выполнится соответствующий *БлокОпераций*;
4. Если *Значение* не найдено, то выполнится *БлокОперацийПоУмолчанию*.

Блок *default* может отсутствовать, тогда если *Значение* не найдено, не выполнится никаких действий.

Опция *break* осуществляет выход из оператора.

Порядок выполнения работы

1. Написать программу сравнения двух введенных с клавиатуры чисел. В результате работы программа должна выводить результат сравнения.
2. Объяснить выбор того или иного условного оператора.

Контрольные вопросы

1. Что такое условный оператор?
2. Какие условные операторы Вы знаете?
3. В каком случае предпочтительнее использовать оператор *if*, а в каком *switch..case*?

ЛАБОРАТОРНАЯ РАБОТА №19

«Язык программирования Си. Операторы цикла»

Цель работы: усвоить синтаксис и особенности использования операторов цикла языка Си.

Введение

Циклом называется блок кода, который для решения задачи требуется повторить несколько раз.

Цикл состоит из блока проверки условий и тела цикла. Цикл выполняется до тех пор, пока значение блока проверки условий истинно. Тело цикла содержит последовательность операций, которая выполняется в случае истинного условия повторения цикла.

В языке Си следующие виды циклов:

while – цикл с предусловием;
do…while – цикл с постусловием;
for – параметрический цикл (цикл с заданным числом повторений).

Цикл с предусловием while

Цикл имеет следующую конструкцию:

```
while (Условие)
{
    БлокОпераций;
}
```

Если *Условие* выполняется (выражение, проверяющее *Условие*, не равно нулю), то выполняется *БлокОпераций*, заключенный в фигурные скобки, затем *Условие* проверяется снова.

Последовательность действий, состоящая из проверки *Условия* и выполнения *БлокаОпераций*, повторяется до тех пор, пока выражение, проверяющее *Условие*, не станет ложным (равным нулю). При этом происходит выход из цикла, и производится выполнение операции, стоящей после оператора цикла.

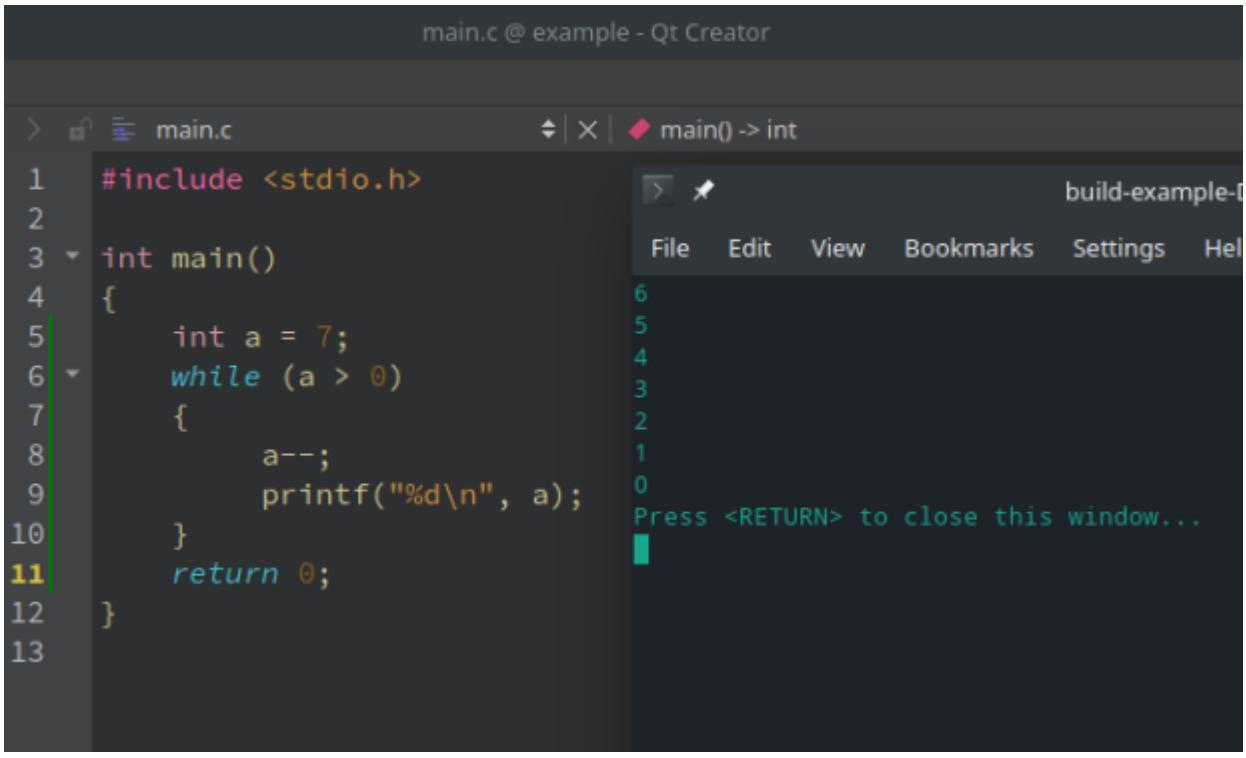

Рис. 19.1. Результат работы цикла *while*

При построении цикла *while*, в него необходимо включить конструкции, изменяющие величину проверяемого выражения так, чтобы в конце концов оно стало ложным (равным нулю). Иначе выполнение цикла будет осуществляться бесконечно (бесконечный цикл). Бесконечный цикл используется при написании программ микроконтроллеров, без бесконечного цикла контроллер остановится после выполнения основной функции.

Пример бесконечного цикла:

```
while (1)
{
    БлокОпераций;
}
```

Цикл с постусловием do..while

Общая форма записи:

```
do {
    БлокОпераций;
} while (Условие);
```

Цикл *do...while* – это цикл с постусловием, где истинность выражения, проверяющего *Условие*, проверяется после выполнения *БлокаОпераций*, заключенного в фигурные скобки. Тело цикла выполняется до тех пор, пока выражение, проверяющее *Условие*, не станет ложным, то есть тело цикла с постусловием выполнится хотя бы один раз.

Использовать цикл *do...while* лучше в тех случаях, когда должна быть выполнена хотя бы одна итерация, либо когда инициализация объектов, участвующих в проверке условия, происходит внутри тела цикла.

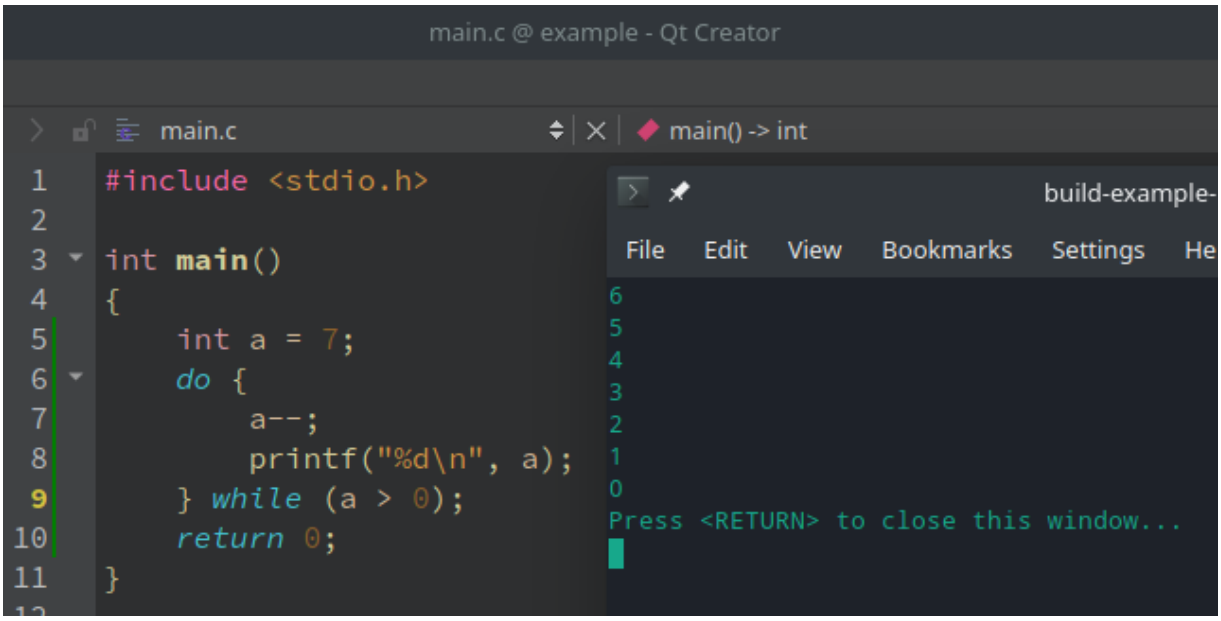

Рис. 19.2. Пример работы цикла *do..while*

Параметрический цикл for

Общая форма записи:

```
for (Инициализация; Условие; Модификация)
{
    БлокОпераций;
}
```

for — параметрический цикл (цикл с фиксированным числом повторений). Для организации такого цикла необходимо осуществить три операции:

1. *Инициализация* – присваивание параметру цикла начального значения;

2. *Условие* – проверка условия повторения цикла, чаще всего – сравнение величины параметра с некоторым граничным значением;

3. *Модификация* – изменение значения параметра для следующего прохождения тела цикла.

Эти три операции записываются в скобках и разделяются точкой с запятой «;». Как правило, параметром цикла является целочисленная переменная.

Инициализация параметра осуществляется только один раз – когда цикл *for* начинает выполняться.

Проверка *Условия* повторения цикла осуществляется перед каждым возможным выполнением тела цикла. Когда выражение, проверяющее *Условие* становится ложным (равным нулю), цикл завершается. Модификация параметра осуществляется в конце каждого выполнения тела цикла. Параметр может увеличиваться или уменьшаться.

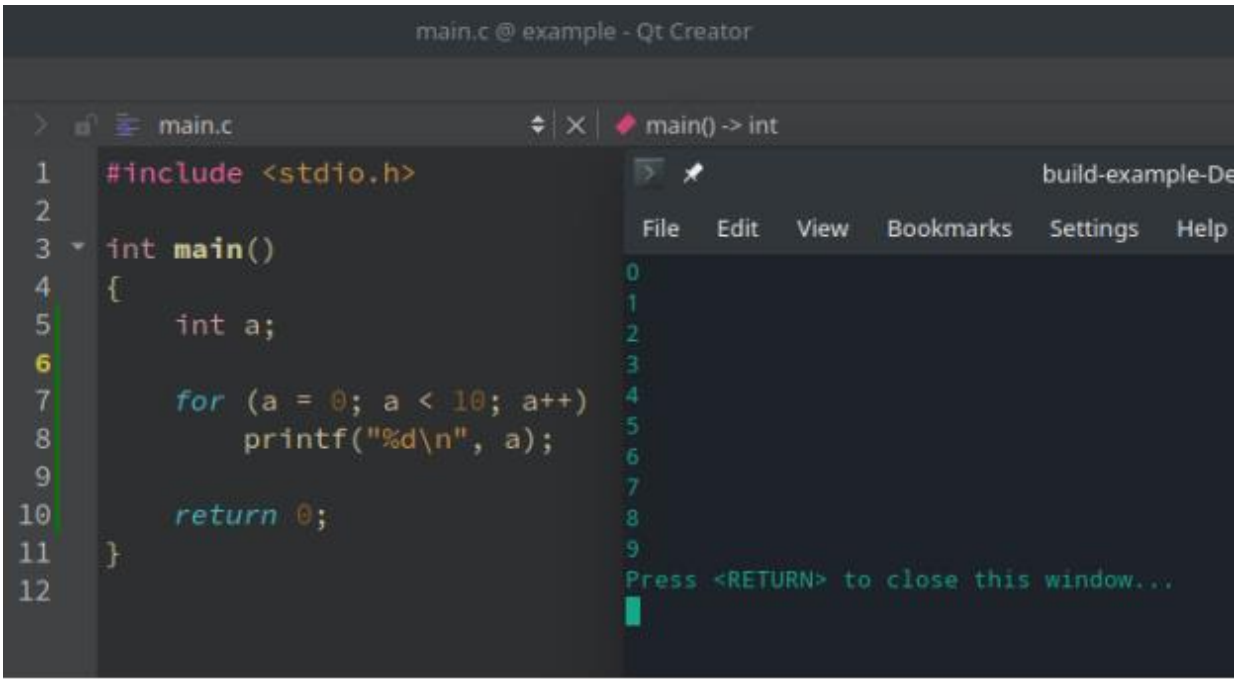

Рис. 19.3. Пример работы цикла *for*

В цикле *for* может использоваться операция запятая для разделения нескольких выражений. Это позволяет включить в спецификацию цикла несколько инициализирующих или корректирующих выражений. Выражения, к которым применяется операция запятая, будут вычисляться слева направо.

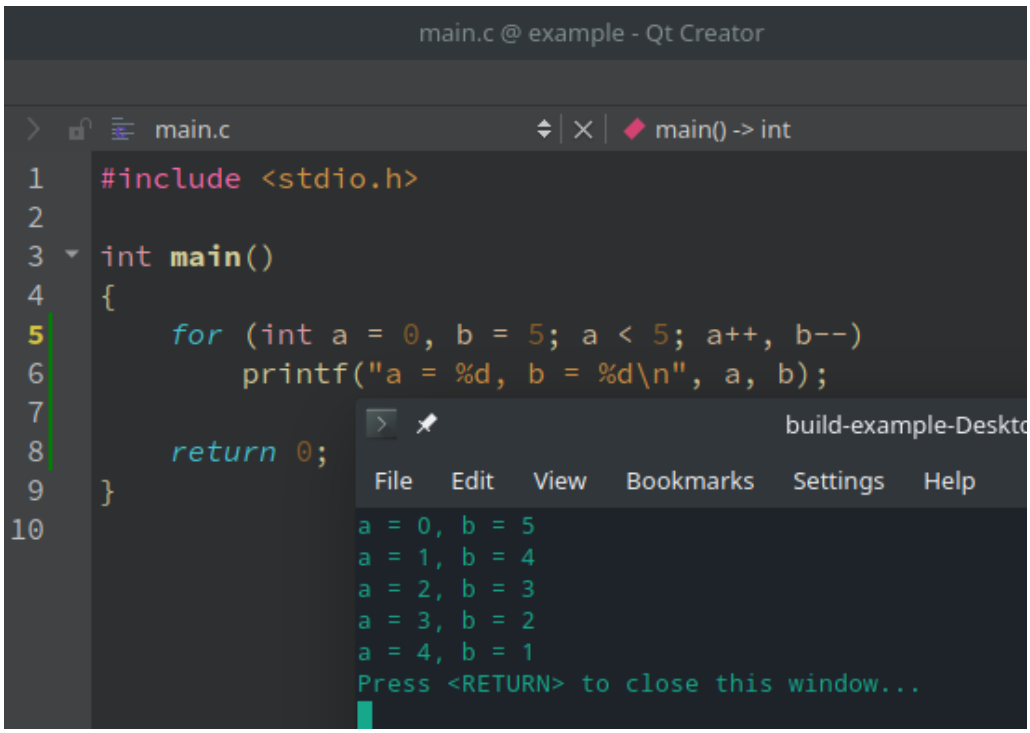

Рис. 19.4. Пример цикла *for* со сложным условием

Обратим внимание, что объявлять переменные можно прямо внутри цикла.

Цикл *for* может быть вложенным аналогично оператору *if*.

В теле любого цикла можно использовать операторы прерывания цикла – *break* и продолжения цикла – *continue*. Оператор *break* позволяет выйти из цикла, не завершая его. Оператор *continue* позволяет пропустить часть операторов тела цикла и начать новую итерацию.

Порядок выполнения работы

1. Написать программу заполнения массива числами, введенными с клавиатуры, используя один из изученных циклов.

Контрольные вопросы

1. Что такое «цикл» в языке программирования?
2. Чем объясняется удобство использования циклов по сравнению с многократным повтором одной и той же операции?
3. Быстрее ли работает программа с использованием цикла, чем такая же, но с постоянным повтором действий?

ЛАБОРАТОРНАЯ РАБОТА №20

«Язык программирования Си. Указатели»

Цель работы: изучить объект «указатель» языка Си, приобрести навыки его использования, освоить операцию взятия адреса.

Введение

Указателем называется переменная, значение которой является адресом другой переменной.

Как говорилось ранее, все переменные располагаются в оперативной памяти. Память – это непрерывная последовательность ячеек. Каждая ячейка имеет свой адрес – порядковый номер, и в нее может быть записан всего один бит информации. Переменная может занимать разное (но кратное 8) количество бит в зависимости от ее типа. *Адресом переменной* является номер ячейки памяти, с которой начинается расположение этой переменной в памяти.

Указатель, являясь переменной, также располагается в оперативной памяти, но его размер фиксирован и зависит от разрядности системы. Для 32-разрядных систем указатель занимает 32 бита или 4 байта памяти.

Указатель объявляется следующим образом:

```
тип *имя;
```

Тип – это тип переменной, адрес которой содержит указатель.

Пример:

```
int a; //объявлена переменная a типа int

int *p; //объявлен указатель с именем p на переменную типа int
```

В приведенном примере указатель *p* пока никак не связан с переменной *a*, так как он не инициализирован. Для инициализации указателя в языке *C* предусмотрена операция взятия адреса, которая обозначается амперсандом &, расположенным перед именем переменной.

```
p = &a; //теперь указатель p содержит адрес переменной a
```

На рисунке 20.1 приведен листинг программы и результат ее работы. В 5-й строке объявлена переменная *a*, в 6-й – указатель *p*, в 7-й строке в указатель *p* записан адрес переменной *a*. В строках 8 и 9 с помощью функции *printf()* и модификатора *%x* выводится двумя способами адрес переменной *a* в шестнадцатеричном виде.

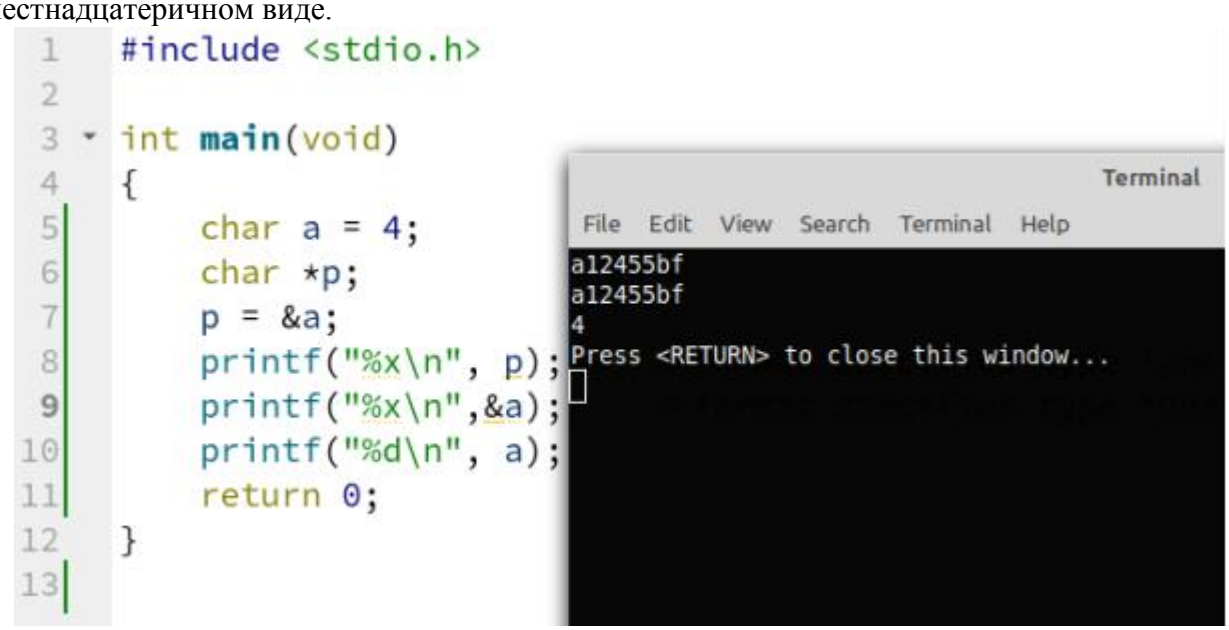

Рис. 20.1. Результат работы программы с использованием указателя

В языке *C* имеется операция получения значения по адресу, которая обозначается звездочкой *.

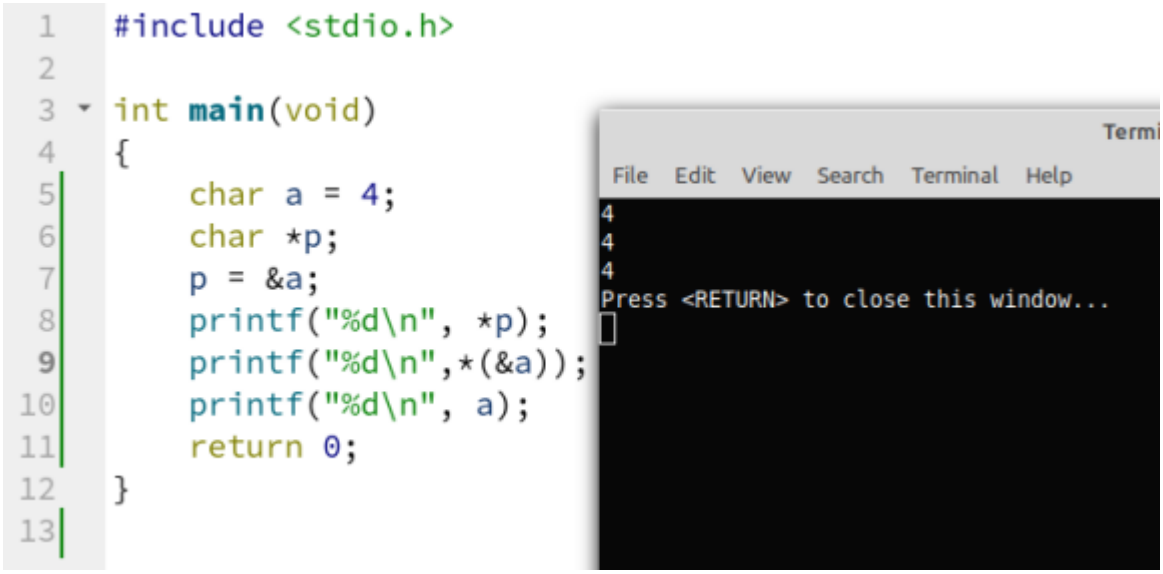

Рис. 20.2. Пример получения адреса переменной

В приведенном листинге в 8-й строке на экран выводится значение по адресу, который содержится в указателе. В 9-й строке это же значение записано в более явном виде.

Имя массива также является указателем, который содержит адрес первого элемента этого массива.

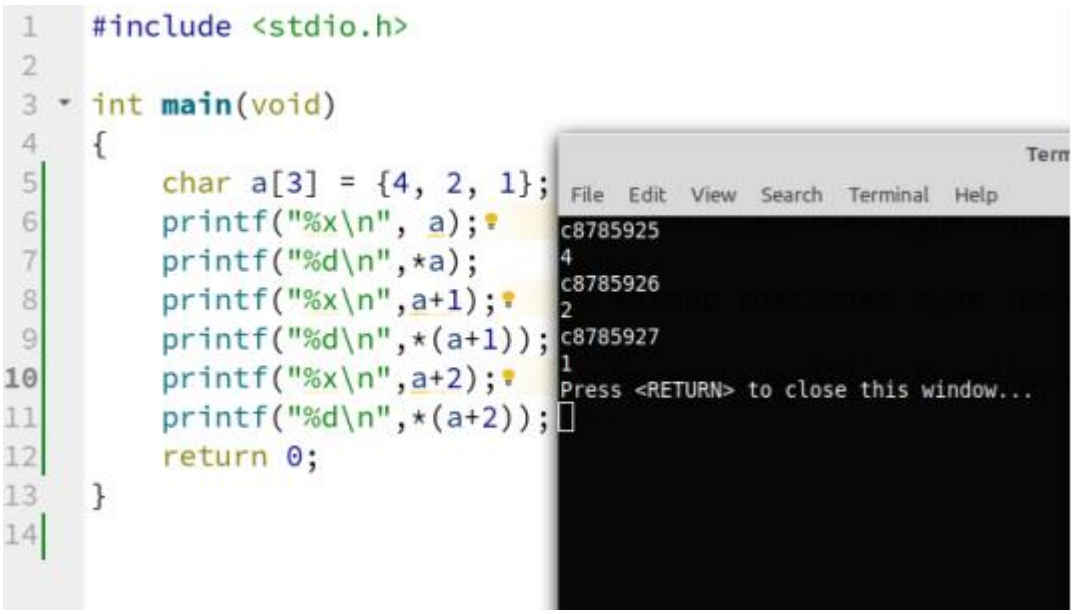

Рис. 20.3. Имя массива является указателем

В данном примере в 6-й строке в качестве аргумента функции *printf()* используется имя массива, и, как видно в результате вывода 7-й строки, оно совпадает с адресом первого элемента массива. В 8-й и 9-й строках к адресу первого элемента прибавили 1. Так как тип массива *char*, каждый его элемент занимает в памяти 1 байт. Прибавление к адресу единицы означает прибавление одного байта. Поэтому в выводе 8-й и 9-й строк содержатся адрес и значение второго элемента массива. Аналогично с 10-й и 11-й строками. Если бы массив имел тип *int*, который занимает в памяти 4 байта, то прибавление единицы соответствовало бы прибавлению к адресу четырех байт, т.е. плюс один размер этого же типа. Это означает, что к указателям применимы арифметические операции.

Порядок выполнения работы

1. Написать программы приведенных выше примеров.
2. Прокомментировать написанные программы. Объяснить алгоритмы их работы.

Контрольные вопросы

1. Что такое «указатель»? Что такое «адрес» переменной?
2. В чем преимущество использования указателей?
3. Как и зачем использовать указатели при передаче агрументов функции?

ЛАБОРАТОРНАЯ РАБОТА №21

«Программирование контроллера *ATmega16*. Настройка портов»

Цель работы: изучить объект «указатель» языка Си, приобрести навыки его использования, освоить операцию взятия адреса.

Введение

После успешной настройки проекта из лабораторной работы №13 и его компиляции без ошибок, можно приступить к написанию программы.

Для начала разберем структуру проекта, изучим основные его файлы и их содержание. Начнем с заголовочного файла *main.h*, расположенного в директории *include* и подключенного в файле *main.c*.

Главным заголовочным файлом, указанным в *main.h*, является заголовок *io.h*, подключение которого выглядит так:

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

В нем содержатся директивы препроцессора, определяющие модель используемого контроллера. Само указание модели расположено в файле проекта *atmega16.config* и выглядит так:

```
#define __AVR_ATmega16__
```

Это указание автоматически подключает в файле *io.h* блок *else* ветвления *if* препроцессора, который с помощью директивы *#include* включает в проект заголовок *iom16.h*:

```
266  #  include <avr/iom32hvbrevb.h>
267  #elif defined (__AVR_ATmega406__)
268  #  include <avr/iom406.h>
269  #elif defined (__AVR_ATmega16__)
270  #  include <avr/iom16.h>
271  #elif defined (__AVR_ATmega16A__)
272  #  include <avr/iom16a.h>
```

Рис. 21.1. Подключение заголовочного файла контроллера

В заголовочном файле *iom16.h* содержится полное описание памяти, портов, регистров, периферии контроллера *ATmega16*. Просмотреть его содержимое можно нажатием на нем левой кнопкой мыши с зажатой клавишей *Ctrl*.

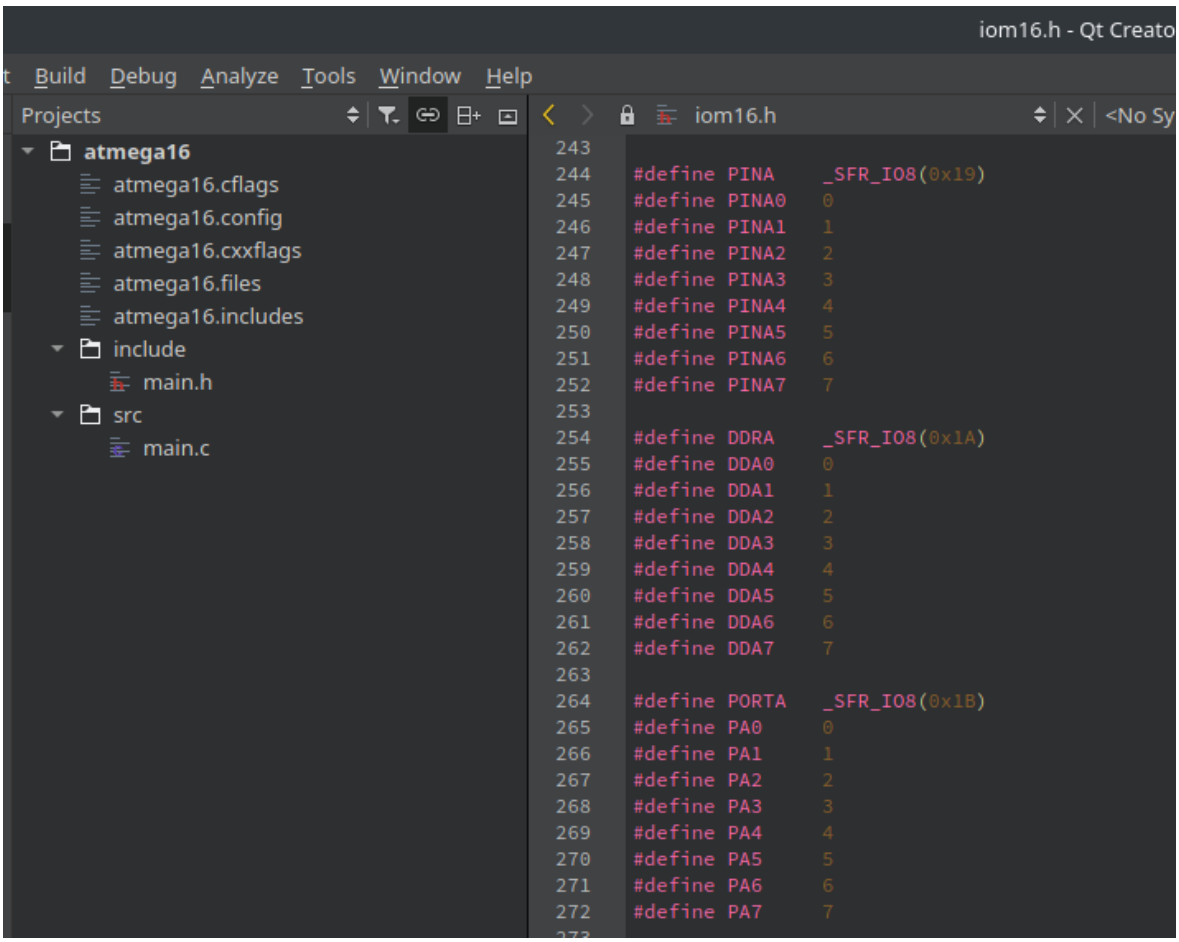

Рис. 21.2. Описание периферии контроллера

Как видно из рисунка 21.2 выше, все имена портов, выводов, регистров и т.д. являются обычными псевдонимами чисел, но эти числа точно соответствуют адресам регистров в памяти или номерам битов в регистрах.

Исправлять содержимое встроенных заголовочных файлов, таких как *iom16.h*, нельзя.

В этой и последующих лабораторных работах будем писать программу контроллера для разработанного ранее устройства.

Переключимся в файл *main.c* и добавим в него функцию настройки периферии контроллера, назовем ее *setup*. В эту функцию добавим настройку портов *C* и *D* на вывод, так как эти порты управляют анодами и катодами 7-сегментного индикатора.

```
void setup()
{
    DDRC = 0xFF;   //настройка порта C на вывод
    DDRD = 0xFF;   //настройка порта D на вывод
}
```

Для удобства объявим в файле *main.h* глобальный массив из десяти элементов, каждый из которых будет содержать комбинации нулей и единиц, соответствующих цифрам на индикаторе.

```
unsigned char digit[10]  =  {0b11000000,   //0
                             0b11111001,   //1
                             0b10100100,   //2
                             0b10110000,   //3
                             0b10011001,   //4
                             0b10010010,   //5
```

```
    0b10000010,   //6
    0b11111000,   //7
    0b10000000,   //8
    0b10010000};  //9
```

Поскольку для отображения информации нами используется динамическая индикация, сигналы управления катодами индикатора от контроллера передаются на все выводы катодов всех разрядов индикатора, а включение определенного разряда определяется сигналом соответствующего анода.

Раздельное управление битами регистров портов контроллера не предусмотрено, чтобы установить в состояние 1 один бит регистра порта, нужно произвести запись всех битов регистр. Чтобы при этом не менять состояние остальных битов порта, можно использовать логическое сложение, при этом 1 запишем в тот разряд, который нужно установить в 1, а в остальные биты - 0. При сложении с 0 состояние бита не изменится, а с 1 - станет равно 1. Для сброса бита в 0, используем тот же принцип, но с операцией умножения.

В разработанной принципиальной схеме устройства катоды десятичных точек индикатора подключены к восьмому биту порта *C*, напишем два макроса для включения и выключения десятичных точек:

```
#define DP_ON PORTC |= 0b10000000;
#define DP_OFF PORTC &= 0b01111111;
```

Теперь при указании в программе команды *DP_ON* текущее состояние порта *C* будет сложено логически с числом 10000000, т.е. 8-й бит порта станет равен 1, а остальные биты останутся без изменения. При *DP_OF* – выполнится логическое умножение, 8-й бит станет равен 0, а остальные не изменятся.

Напишем аналогичные макросы для включения и выключения разрядов индикатора, согласно подключению на принципиальной схеме:

```
#define DIG1_ON PORTD |= 0b00000001;
#define DIG1_OFF PORTD &= 0b11111110;
#define DIG2_ON PORTD |= 0b00000010;
#define DIG2_OFF PORTD &= 0b11111101;
#define DIG3_ON PORTD |= 0b00000100;
#define DIG3_OFF PORTD &= 0b11111011;
```

Напишем в файле *main.c* функцию для записи чисел в индикатор:

```
void showDigit(unsigned char digitNumber, unsigned char value)
{
    PORTC = digit[value];
    if (digitNumber == 1) {DIG1_ON; DIG2_OFF; DIG3_OFF;}
    if (digitNumber == 2) {DIG1_OFF; DIG2_ON; DIG3_OFF;}
    if (digitNumber == 3) {DIG1_OFF; DIG2_OFF; DIG3_ON;}
}
```

Функция ничего не возвращает в качестве результата, поэтому ее тип *void*, и в ней отсутствует оператор *return*. В качестве аргументов функции указан номер разряда, считая с 1 слева, и цифра для отображения.

Теперь в программе можно вызывать в любом месте функцию *showDigit* и на индикаторе в указанном разряде будет отображена указанная цифра. Например,

```
    showDigit(1, 4);
```

в самом левом разряде будет выведена цифра 4.

Порядок выполнения работы

1. Открыть шаблон проекта из ЛР №13.
2. Написать программу конфигурирования портов контроллера для выбранного ранее устройства.

Контрольные вопросы

1. Что такое «порт» микроконтроллера?
2. Объясните алгоритм настройки портов *ATmega16*?
3. Какие регистры в *ATmega16* имеются для настройки портов?

ЛАБОРАТОРНАЯ РАБОТА №22

«Динамическая индикация»

Цель работы: получить навыки организации динамической индикации при отображении информации.

Введение

В предыдущей работе мы создали функцию *showDigit*, аргументами которой являются номер разряда индикатора и отображаемая на нем цифра.

Попробуем теперь вывести разные цифры сразу во все три разряда индикатора, например, так:

```
showDigit(1, 3);
showDigit(2, 5);
showDigit(3, 4);
```

В результате на индикаторе будет цифра 4 в последнем разряде, при этом первые два разряда будут выключены - таков алгоритм функции *showDigit* при динамической индикации.

Рис. 22.1. Результат работы функции вывода информации

Чтобы информация отображалась одновременно в трех сегментах индикатора, нужно достаточно быстро выводить цифры в разные разряды, для этого можно поместить все три вышеуказанные строки в вечный цикл контроллера.

```
while (1)
{
    showDigit(1, 3);
    showDigit(2, 5);
    showDigit(3, 4);
}
```

Результат после компиляции и загрузки:

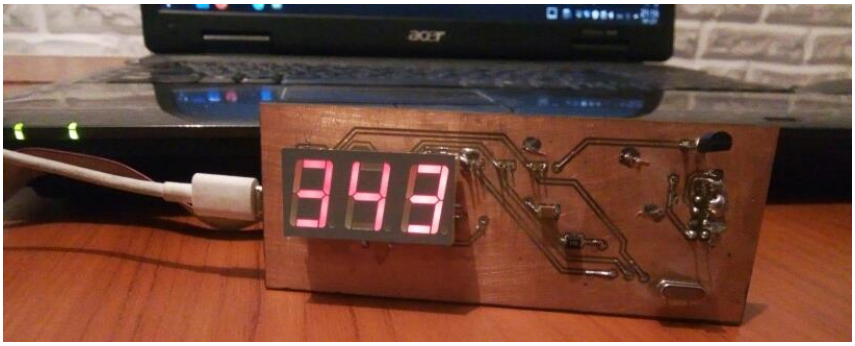

Рис. 22.2. Вывод информации в разные разряды индикатора в цикле

На индикаторе отображены не те цифры, т.к. индикатор попросту «не успевает» за контроллером, работающем на частоте 16 миллионов тактовых импульсов в секунду. Добавим задержку после каждой команды *showDigit*.

Для организации задержки воспользуемся стандартной функцией *_delay_ms(n)*, аргументом n которой является время в миллисекундах. Достаточно задержки 5 мс.

Функция *_delay_ms* может быть доступна для использования при подключении заголовочного файла:

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

Эта строка подключения уже содержится в проекте в файле *main.h*, дополнительные подключения не требуются.

Результат добавления задержки:

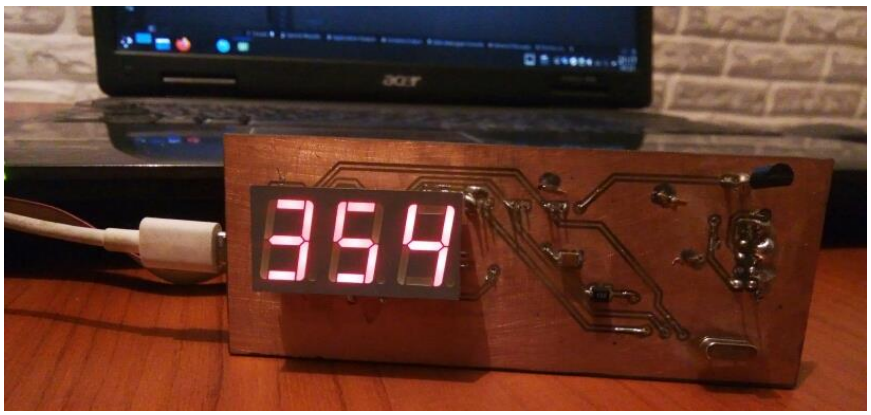

Рис. 22.3. Результат добавления задержки вывода информации

Функция *_delay_ms* использует тактирующий сигнал контроллера, поэтому задержка, организованная этой функцией, является очень точной.

Выводить информацию в каждый разряд отдельно, добавляя каждый раз задержку, не очень удобно. Было бы лучше написать функцию, аргумент которой не цифра, а целое число, содержащее от одного до трех разрядов, а функция бы сама разделяла число на цифры и выводила каждую из них в соответствующий разряд.

Объявим функцию *writeToDisp*, с целочисленным аргументом:

```
void writeToDisp(int data)
{

}
```

В функции объявим и инициализируем нулями массив типа *char* из трех элементов:

```
char str[3] = "000";
```

В этом массиве будут храниться коды цифр. Воспользуемся функцией *sprintf* языка Си для конвертации целого числа в символы. Формат функции следующий:

```
int sprintf( char *buffer, const char *format, ... )
```

Функция *sprintf()* идентична функции *printf()* за исключением того, что информация записывается в массив, адресуемый указателем *buf*, а не в стандартный поток (например, на экран). Прототип функции (т.е. ее заголовок) находится в файле *stdio.h*, подключим его с помощью *#include* в начале файла *main.h*.

Добавим в функцию *writeToDisp* строку:

```
sprintf(str, "%.3d", data);
```

Модификатор *.3d* сообщает функции, что результат записи в массив должен содержать 3 разряда целого числа. Если число имеет меньшее количество разрядов, недостающие разряды будут заполнены нулями.

Теперь добавим созданную ранее функцию *showDigit*, в качестве второго аргумента которой нужно указать число, которое соответствует цифре разряда, код которой содержится в элементе массива *str*. Здесь возникает несоответствие: функция *showDigit* должна получить вторым аргументом целое число от 0 до 9, а передать ей мы можем *ascii*-код символа, ведь именно он содержится в элементах массива. Обратимся к таблице *ascii* символов:

Decimal	Hexadecimal	Binary	Octal	Char
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9

Рис. 22.4. Таблица ASCII

Как видно из таблицы, цифра 0 имеет десятичный код 48, 1 - 49 ... 9 - 57. Нужно понимать разницу между числом, обозначающим количественную меру чего-либо, и цифрой, являющейся символическим обозначением числа в некоторой системе обозначений. Чтобы получить число из символа в языке Си, вычтем алгебраически символы, которые принято указывать в одинарных кавычках. Например:

```
'6' - '0'
```

Эта запись эквивалентна вычитанию 54 - 48, результат которого 6. Т.е. если нужно получить число из символа, нужно из этого символа вычесть *ascii*-код нуля.

Добавим в функцию *writeToDisp* строки:

```
showDigit(1, str[0] - '0');
_delay_ms(5);
showDigit(2, str[1] - '0');
_delay_ms(5);
showDigit(3, str[2] - '0');
_delay_ms(5);
```

Теперь в вечный цикл основной программы можно написать всего одну строку с функцией *writeToDisp(n)*, где *n* - целое положительное число для записи в индикатор.

ЛАБОРАТОРНАЯ РАБОТА №23

«Работа с АЦП контроллера *ATmega16*»

Цель работы: усвоить назначение объекта «структура», изучить способы объявления структур и инициализации их полей.

Введение

Для измерения температуры окружающего воздуха в состав устройства входит датчик температуры *KTY81/120*, подключенный к каналу 0 аналогово-цифрового преобразователя *ATmega16*.

Преобразователь контроллера представляет собой АЦП последовательного приближения, имеет 8 независимых каналов и обеспечивает 10-битный результат.

Для работы с АЦП в *ATmega16* предусмотрены специальные регистры, которые должны быть настроены перед использованием АЦП. Прежде всего, необходимо "сообщить" контроллеру тип источника опорного напряжения преобразователя. При проектировании принципиальной схемы нами было выбрано подключение вывода *AREF* на "землю" через конденсатор.

В документации контроллера на стр. 217 представлен один из регистров настройки АЦП, который называется *ADMUX*.

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 23.1. Регистр *ADMUX*

Шестой и седьмой биты этого регистра (*REFS0* и *REFS1*) предназначены для выбора источника опорного напряжения.

Table 83. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Рис. 23.2. Назначение битов регистра *ADMUX*

Как видно из таблицы, для нашего случая подходит вторая строка, т.е. используется опорное напряжение с вывода *AVCC*, при этом вывод *AREF* подключен через конденсатор.

Пятый бит регистра *ADMUX* называется *ADLAR* (*ADC Left Adjust Result*) и предназначен для выравнивания результата по левому или правому краю регистров результата. Результат преобразования всегда помещается и хранится в двух 8-битных регистрах *ADCH* и *ADCL*, представляющих собой старший и младший байт результата. Однако результат преобразования состоит из 10 бит, а значит, он может быть расположен в этих двух байтах по-разному:

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Рис. 23.3. Бит *ADLAR*

По умолчанию бит *ADLAR* равен нулю, т.е. 8 бит результата помещаются в младший байт *ADCL*, а два оставшихся бита - в старший *ADCH*. Общий результат преобразования может быть доступен в программе или по именам *ADCH* и *ADCL*, или по одному имени *ADC*.

Биты *MUX4..0* предназначены для выбора канала и способа преобразования. В самом простом случае, когда входной сигнал подключен к одному выводу контроллера, биты *MUX4* и *MUX3* должны быть равны нулю, а битами *MUX2, MUX1, MUX0* выбирают канал от 000 (канал 0) до 111 (канал 7). Итого 8 каналов. АЦП *ATmega16* может работать в так называемом дифференциальном режиме, когда измерение напряжения происходит не между каналом и общей точкой, а между двумя выводами контроллера. Для этих целей используются биты *MUX4* и *MUX3*.

В нашем случае преобразование простое, с одного канала 0, поэтому все биты *MUX4..0* должны быть равны нулю.

Настроить регистр *ADMUX* можно так:

```
ADMUX = 0x40;
```

Однако информативнее будет произвести запись в двоичной системе:

```
ADMUX = 0b01000000;
```

При таком способе записи видно, чему равен каждый бит регистра.
В среде разработчиков более популярен иной способ записи битов в регистр:

```
ADMUX |= (1 << RFS0);
```

В записи выше логическая единица сдвигается влево на 6 разрядов. Именованная константа *RFS0* объявлена с помощью *#define* в файле *iom16.h* и соответствует числу 6 в *ATmega16*.

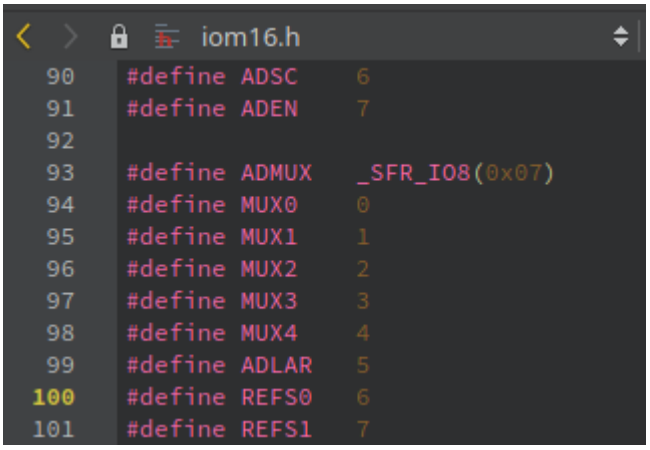

Рис. 23.4. Часть листинга файла *iom16.h*

Затем состояние регистра *ADMUX* логически складывается со сдвинутой на 6 разрядов влево единицей (0b01000000). Результат логического сложения записывается снова в регистр *ADMUX*.

Такой способ записи наглядно демонстрирует, какой бит изменяется, а самое главное – обеспечивает универсальность программы.

В разных моделях контроллеров *AVR* биты могут располагаться в разном порядке. Конфигурируя регистр в явном виде, т.е. конкретным числом, мы делаем программу пригодной только для конкретного контроллера, и вряд ли она будет верно работать в другой модели. Встроенные имена битов, такие как *RFS0*, с большей вероятностью будут встречаться и в других моделях контроллеров *AVR*, пусть и соответствуя другим номерам битов.

Вторым специальным регистром для работы с АЦП является регистр *ADCSRA* - регистр управления и состояния АЦП.

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

Рис. 23.5. Регистр *ADCSRA*

Бит 7 - *ADEN* - разрешение работы АЦП.
Бит 6 - *ADSC* - старт единичного преобразования.
Бит 5 - *ADATE* - разрешение автозапуска преобразования по переднему фронту преобразуемого входного сигнала.

Бит 4 - *ADIF* - признак (флаг) прерывания автоматически устанавливается в 1 по окончании преобразования.

Бит 3 - *ADIE* - разрешение прерывания от преобразователя.

Биты 2..0 - *ADPS2..0* - биты выбора делителя частоты преобразователя определяют скорость работы преобразователя.

Table 85. ADC Prescaler Selections

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

Рис. 23.6. Предделитель АЦП

Добавим следующую строку в функцию *setup()*:

```
ADCSRA |= (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);
```

Работа АЦП будет разрешена, частота работы будет в 128 раз меньше частоты тактового сигнала, прерывания от АЦП отключены.

Следующим шагом работы с АЦП является написание функции, запускающей преобразование выбранного канала и возвращающей результат преобразования.

```
unsigned int readADC(unsigned char channel)
{
    channel &= 0b00000111;
    ADCSRA |= (1 << ADSC);
    while (!(ADCSRA & (1<<ADIF)));
    return (ADC);
}
```

Возвращаемое функцией значение – целое положительное 16-битное число. Аргумент функции, указанный в ее заголовке в скобках - номер преобразуемого канала АЦП.

В первой строке полученный функцией номер канала логически умножается на число 7 (0b00000111), чтобы ограничить возможные ошибки при указании номера канала больше 7.

Во второй строке выполняется запуск преобразования записью 1 в бит *ADSC* регистра управления и состояния АЦП *ADCSRA*.

Третья строка обеспечивает паузу выполнения программы на время работы АЦП. Читается строка так: "повторять проверку условия пока результат логического умножения состояние регистра *ADCSRA* и сдвинутой влево на 4 разряда единицы (бит *ADIF* четвертый по номеру) равен нулю". Программа будет повторять цикл *while* до тех пор, пока условие цикла ложно, т.е. пока флаг готовности *ADIF* не станет равен 1.

В последней строке возвращается результат преобразования – число, пропорциональное напряжению на 37-м выводе контроллера.

Порядок выполнения работы

1. Написать функции настройки АЦП для разрабатываемого устройства.

Контрольные вопросы

1. Опишите процесс настройки регистров АЦП контроллера *ATmega16*.
2. Что такое предделитель АЦП?

ЛАБОРАТОРНАЯ РАБОТА №24

«Обработка результатов работы АЦП»

Цель работы: получить навыки работы с результатами аналого-цифрового преобразования и приведения их к соответствующим физическим величинам.

Введение

Ранее была произведена настройка АЦП, а также создана функция запуска преобразования и получения результата, представляющего собой число, пропорциональное входному напряжению. К настоящему моменту функция настройки контроллера выглядит так:

```
void setup()
{
    DDRC = 0xFF;   //настройка порта C на вывод
    DDRD = 0xFF;   //настройка порта D на вывод
    ADMUX = (1<<REFS0); //выбор источника опорного напряжения АЦП
    ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0); //включение АЦП и настройка делителя частоты
}
```

Написаны функции формирования цифры на индикаторе и вывода числа на экран:

```
void showDigit(unsigned char digitNumber, unsigned char value)
{
    PORTC = digit[value];
    if (digitNumber == 1) {DIG1_ON; DIG2_OFF; DIG3_OFF;}
    if (digitNumber == 2) {DIG1_OFF; DIG2_ON; DIG3_OFF;}
    if (digitNumber == 3) {DIG1_OFF; DIG2_OFF; DIG3_ON;}
}

void writeToDisp(unsigned int data)
{
    char str[3] = "000";
    sprintf(str, "%.3d", data);
    showDigit(1, str[0] - '0');
    _delay_ms(5);
    showDigit(2, str[1] - '0');
    _delay_ms(5);
    showDigit(3, str[2] - '0');
    _delay_ms(5);
}
```

Написана функция запуска преобразования и получения результата:

```
unsigned int readADC(unsigned char channel)
{
    channel &= 0b00000111;
    ADMUX = 0x40;
    ADCSRA |= (1<<ADSC);
    while (!(ADCSRA & (1<<ADIF)));
    return (ADC);
}
```

Теперь напишем функцию, преобразующую полученный от АЦП результат в температуру. Функция будет возвращать результат в градусах Цельсия, а в качестве аргумента в скобках принимать результат АЦП.

```
int getTemp(unsigned int adcResult)
{
```

Вычисления будем производить последовательно: сначала пересчитаем результат АЦП в напряжение, затем из полученного напряжения вычислим сопротивление датчика температуры в момент преобразования и по сопротивлению вычислим температуру.

Объявим переменные типа *float*, т.к. их значения дробные:

```
float volt, resist, t;
```

Вычислим напряжение:

```
volt = 5*adcResult/1024; //расчет напряжения приведен в лекции
```

Так как датчик температуры подключен в нижнем плече делителя напряжения (см. лекцию), его сопротивление вычисляется по выражению:

$$R2 = R1*Uвых/(Uвх - Uвых)$$

Напишем строку для вычисления сопротивления согласно этому выражению:

```
    resist = 2200*volt/(5 - volt);
```

В написанной строке 2200 - это сопротивление резистора *R1* верхнего плеча делителя напряжения в Ом, 5 – входное напряжение делителя в Вольтах, равное напряжению питания схемы.

Для вычисления температуры обратимся к документации датчика *KTY81/120*, в которой присутствует таблица соответствия сопротивления датчика его температуре:

Table 7. Ambient temperature, corresponding resistance, temperature coefficient and maximum expected temperature error for KTY81/110 and KTY81/120

$I_{sen(cont)}$ = 1 mA.

Ambient temperature		Temperature coefficient (%/K)	KTY81/110				KTY81/120			
(°C)	(°F)		Resistance (Ω)			Temperature error (K)	Resistance (Ω)			Temperature error (K)
			Min	Typ	Max		Min	Typ	Max	
−55	−67	0.99	475	490	505	±3.02	470	490	510	±4.02
−50	−58	0.98	500	515	530	±2.92	495	515	535	±3.94
−40	−40	0.96	552	567	582	±2.74	547	567	588	±3.78
−30	−22	0.93	609	624	638	±2.55	603	624	645	±3.62
−20	−4	0.91	669	684	698	±2.35	662	684	705	±3.45
−10	14	0.88	733	747	761	±2.14	726	747	769	±3.27
0	32	0.85	802	815	828	±1.91	793	815	836	±3.08
10	50	0.83	874	886	898	±1.67	865	886	907	±2.88
20	68	0.80	950	961	972	±1.41	941	961	982	±2.66
25	77	0.79	990	1000	1010	±1.27	980	1000	1020	±2.54
30	86	0.78	1029	1040	1051	±1.39	1018	1040	1061	±2.68
40	104	0.75	1108	1122	1136	±1.64	1097	1122	1147	±2.97
50	122	0.73	1192	1209	1225	±1.91	1180	1209	1237	±3.28
60	140	0.71	1278	1299	1319	±2.19	1265	1299	1332	±3.61
70	158	0.69	1369	1392	1416	±2.49	1355	1392	1430	±3.94
80	176	0.67	1462	1490	1518	±2.8	1447	1490	1532	±4.3
90	194	0.65	1559	1591	1623	±3.12	1543	1591	1639	±4.66
100	212	0.63	1659	1696	1733	±3.46	1642	1696	1750	±5.05
110	230	0.61	1762	1805	1847	±3.83	1744	1805	1865	±5.48
120	248	0.58	1867	1915	1963	±4.33	1848	1915	1982	±6.07
125	257	0.55	1919	1970	2020	±4.66	1899	1970	2040	±6.47
130	266	0.52	1970	2023	2077	±5.07	1950	2023	2097	±6.98
140	284	0.45	2065	2124	2184	±6.28	2043	2124	2205	±8.51
150	302	0.35	2145	2211	2277	±8.55	2123	2211	2299	±11.43

Рис. 24.1. Зависимость сопротивления датчика от его температуры

Запишем столбцы температуры (в гр. *C*) и сопротивления (столбец *Resistance Typ.*) в *MS Excel*, построим график зависимости температуры от сопротивления, добавим линию тренда и ее выражение:

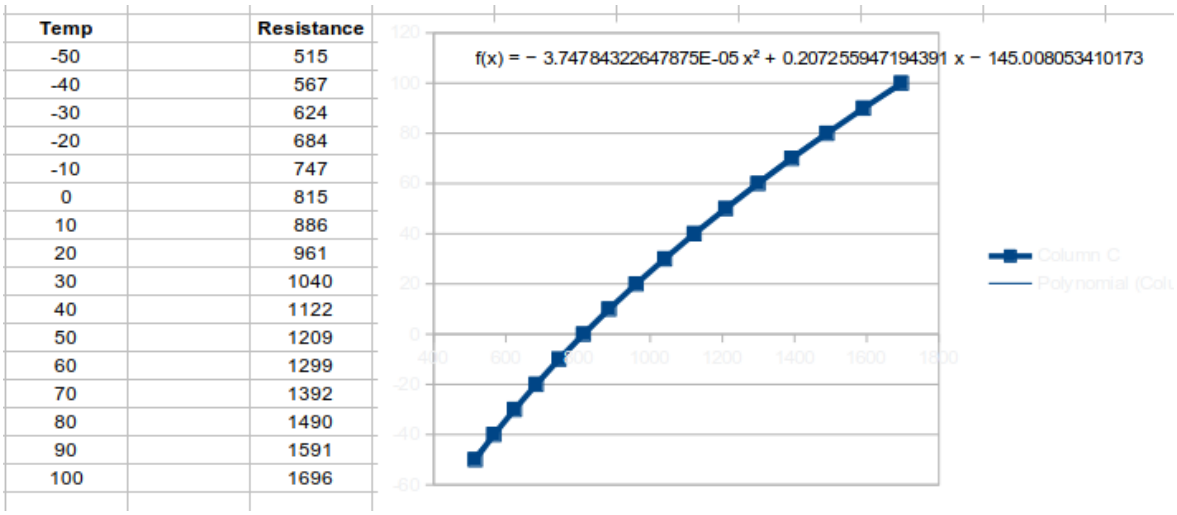

Рис. 24.2. Использование *MS Excel* для получения выражения

Из полученного выражения добавим строку в программу:

```
t = -3.7478*resist*resist*0.00001 + 0.2073*resist - 145.008;
```

Вернем значение переменной t и завершим написание функции:

```
    return (int)t;
}
```

В строке выше используется операция приведения типов. Дело в том, что в переменной t после вычисления будет храниться целая и дробная часть значения температуры. Для разрабатываемого устройства такая точность избыточна, дробную часть можно отбросить. Если опустить указание (*int*), то в процессе компиляции возникнет ошибка, т.к. функция должна вернуть целое число (перед ее именем указан *int*), а в операторе *return* возвращается дробное число типа *float*.

Теперь для вывода температуры на индикатор в основной функции *main* до вечного цикла можно указать:

```
writeToDisp(getTemp(readADC(0)));
```

В функцию *writeToDisp* в качестве аргумента подставляется результат выполнения функции *getTemp*, аргументом которой в свою очередь является результат работы АЦП.

Рис. 24.3. Результат вывода температуры на экран

Порядок выполнения работы

1. Для разрабатываемого устройства написать функции приведения результата преобразования к соответствующим физическим величинам.

Контрольные вопросы

1. Объяснить принцип обработки результатов работы АЦП.
2. Объяснить требования точности вычислений значений физических величин по результатам преобразования.

ЛАБОРАТОРНАЯ РАБОТА №25

«Работа с таймерами *ATmega16*»

Цель работы: получить навыки настройки таймеров контроллера *ATmega16*, а также написания функций обработки их прерываний.

Введение

Ранее была написана функция *getTemp(unsigned int adcResult)* для вычисления температуры воздуха из значения, полученного в результате работы АЦП. Функция была вызвана в основной функции *main* до вечного цикла *while (1)*, а значит, измерение и вывод температуры на индикатор произойдет всего один раз, для обновления показания температуры контроллер нужно перезагрузить. Такой алгоритм может использоваться только на время написания программы и ее отладки. В законченном устройстве температура должна обновляться периодически.

Существует два варианта обновления температуры:

1. В вечном цикле *while (1)*;
2. С помощью таймера/счетчика.

В первом случае при малом объеме кода в цикле *while* обновление будет происходить слишком часто. Из-за высокой скорости работы АЦП при граничных значениях реальной температуры воздуха на индикаторе значения будут меняться постоянно. Например, температура воздуха 25,999 гр. С через несколько мгновений изменится до 26,001 гр. С. Причем физика датчика такова, что его сопротивление постоянно меняется в некоторых пределах, поэтому в его документации для каждого значения температуры приведен диапазон значений сопротивления. Контроллер оцифрует каждое изменение сопротивления. Из вышесказанного следует, что измерение температуры следует производить реже.

Если бы объем программного кода цикла *while* был достаточно большим, то измерение могло бы происходить слишком медленно, т.к. в контроллер выполняет команды цикла последовательно друг за другом. Такой подход неверен, особенно в тех в случаях, когда нужно точно отслеживать изменения аналогового сигнала.

Наиболее правильным решением при измерении температуры является использование таймера/счетчика и его прерывания. Достоинства этого способа в том, что таймер/счетчик является самостоятельным устройством, он работает независимо от процессора контроллера. Пока контроллер выполняет строки программы одну за другой, таймер/счетчик продолжает считать тактовые импульсы, поступающие на его вход.

Чтобы организовать выполнение каких-либо действий контроллера при, например, достижении таймером/счетчиком некоторого значения или при совершении полного цикла счета, следует настроить прерывание по таймеру/счетчику. Система прерываний описана в лекции, а общие принципы настройки таймеров/счетчиков в лекции.

Как и с любой другой периферией контроллера, перед использованием таймеров/счетчиков их нужно настроить. Выберем для работы, например, таймер/счетчик *T1*. Это 16-разрядный таймер/счетчик, т.е. его счетный регистр *TCNT1*, в котором и будет содержаться результат счета, состоит из двух байт.

Любой таймер/счетчик может работать в различных режимах. Для поставленной задачи будем использовать прерывание по переполнению таймера.

Как известно, любой счетчик в режиме прямого счета считает от 0 до максимального своего значения, затем снова переходит к 0. В этот момент и возникает переполнение, т.е. по завершении полного цикла счета. Таймер/счетчик *T1 ATmega16* будет считать от 0000 0000 0000 0000 до 1111 1111 1111 1111 (65 535 в десятичной системе).

Контроллер работает на частоте 16 МГц, период одного импульса таймера/счетчика *T1*, если его тактировать такой частотой, составит 0,0000000625 сек. Умножим это число на количество значений таймера/счетчика *T1*, получим время его полного цикла счета, которое составит 0,004096 сек. Разделим 1 сек на 0,004096, получим 244 полных циклов счета таймера/счетчика за 1 секунду.

Т.е. за 1 секунду прерывание таймера возникнет 244 раза. Это избыточное количество прерываний для измерения температуры.

Уменьшить скорость работы таймера/счетчика можно, включив в работу предделитель частоты. Выберем максимальное значение предделителя *ATmega16*, которое составляет 1024. Тогда частота импульсов на входе таймера/счетчика составит 15 625 Гц. Длительность периода импульса в этом случае будет равна 1 / 15625 = 0,000064 сек. Полный цикл счета в этом случае будет равен 0,000064 * 65 536 = 4,1943 сек.

Будем измерять температуру 1 раз в секунду. Для этого запишем в регистр *TCNT1* таймера/счетчика такое начальное значение, чтобы время счета с этого значения до переполнения составило 1 секунду. Вычислим это значение: 65 536 - 15 625 = 49 911.

Приступим к написанию программы. В функцию *setup()* добавим настройку таймера/счетчика:

```
TCNT1 = 49911;     // начальное значение счетчика
```

Таймер/счетчик *T1* имеет несколько регистров настройки. В регистре *TCCR1A* настраивается поведение таймера в режиме сравнения его значения с заданным, управление внешним выводом контроллера, связанным с *T1*. Для выбранного выше режима работы, эти настройки не нужны, поэтому запишем в этот регистр нули:

```
TCCR1A = 0x00;
```

В регистре *TCCR1B* первыми битами настраивается предделитель. Настроим эти биты:

```
TCCR1B |= (1<<CS10) | (1<<CS12);   //деление частоты в 1024 раза
```

Разрешим работу прерывания таймера/счетчика *T1* по переполнению:

```
TIMSK |= (1 << TOIE1) ;     //включение бита TOIE1 для прерывания
```

Разрешим работу прерываний контроллера:

```
sei(); // Включение глобальных прерываний
```

Напишем функцию обработки прерывания по переполнению регистра *TCNT1* таймера/счетчика *T1*:

```
ISR(TIMER1_OVF_vect)
{
    TCNT1 = 49911;
```

Теперь при возникновении прерывания, контроллер приостановит выполнение цикла *while* и перейдет в функцию прерывания таймера. В этой функции в счетный регистр таймера *T1* снова запишется значение 49 911, и таймер продолжит счет с него.

Вызовем функцию *getTemp* для запуска АЦП и получения значения температуры. Результат выполнения функции поместим в глобальную переменную *int temp*, для этого объявим ее в файле *main.h*. Без глобальной переменной в данном случае не обойтись, т.к. доступ к ней должен выполняться из основной программы и из прерывания, но передать переменную в прерывание в качестве аргумента нельзя. Закончим написание функции прерывания:

```
    temp = getTemp(readADC(0));
}
```

Теперь основная функция *main* будет выглядеть так:

```
int main()
{
    setup();
    while (1)
    {
        writeToDisp(temp);
    }
    return 0;
}
```

В цикле *while* будет постоянно происходить вывод значения переменной *temp* на индикатор, значение этой переменной будет изменяться при каждом прерывании от таймера/счетчика *T1*, которое будет происходить 1 раз в секунду.

Порядок выполнения работы

1. Для разрабатываемого устройства написать функции настройки таймера.
2. Организовать запуск аналого-цифрового преобразования по прерыванию таймера.

Контрольные вопросы

1. Что такое «таймер» контроллера?
2. Какие возможности предоставляют таймеры контроллера *ATmega16*?
3. В чем преимущество использования таймеров по сравнению с циклами?