Кафедра Автоматизации Технологических Процессов

Методические указания к курсу «Микропроцессоры в системах управления»

Тарасов Олег Владимирович

Южанин Виктор Владимирович

Москва 2011

Обзор архитектуры микроконтроллера

Прежде, чем начать подробное рассмотрение архитектуры контроллеров фирмы Atmel, нужно разобраться, что представляют собой микроконтроллеры в целом.

Основное отличие микроконтроллеров от привычных для нас микропроцессоров, установленных в домашних компьютерах, заключается в том, что микроконтроллер — это высокоинтегрированное устройство. Микропроцессоры представляют собой вычислительное устройство, требующее для своей работы присутствия таких внешних компонентов, как загрузочное ПЗУ (ROM), оперативная память (RAM), устройство хранения внешних программ (жёсткий диск) и т. д. Микроконтроллер является однокристальным устройством, в котором присутствует не только сам микропроцессор, но и вся периферия, необходимая для его работы. Для простоты можно провести аналогию между микроконтроллером и обычным компьютером: всё, что мы привыкли видеть в среднестатистическом домашнем компьютере (процессор, оперативная память, жёсткий диск, BIOS), расположено в одной-единственной микросхеме контроллера. Конечно, производительность этих двух систем несопоставима. Однако небольшие вычислительные мощности микроконтроллеров вкупе с их небольшими размерами и низким энергопотреблением находят широчайшее применение в промышленности. Они применяются в самых разнообразных сферах: от измерительных приборов, фотоаппаратов и видеокамер, принтеров, сканеров и копировальных аппаратов до изделий электронных развлечений и всевозможной домашней техники, в результате чего они получили нарицательное название «процессор для стиральных машин».

Архитектура RISC

Все 8-разрядные микроконтроллеры фирмы Atmel, которые мы будем рассматривать в этом курсе, построены на основе производительной архитектуры RISC. Давайте разберёмся, что означает эта аббревиатура.

Все микропроцессоры первого поколения обладали жёстко «прошитой» логикой декодирования команд. В то время технология создания запоминающих устройств сильно «хромала» по сравнению с технологией изготовления процессоров, то есть, более скоростным процессорам приходилось долго ожидать считывания следующей команды для декодирования из медленнодействующего запоминающего устройства. Поэтому, с целью эффективного использования этого периода ожидания, возникла идея заполнить время до поступления следующей команды. В результате были разработаны сложные и комплексные машинные команды. Возникли команды, которые выполняли последовательно несколько внутрипроцессорных операций до тех пор, пока не поступала следующая внешняя команда. Так появились процессоры CISC (Complex Instruction Set Computer — компьютер со сложным набором команд). Типичными представителями этой архитектуры процессоров являются семейства 80x86 и процессор Pentium компании Intel или семейства 680x0 компании Motorola.

Практически все процессоры CISC работают по принципу микропрограммирования, то есть, каждая машинная команда последовательно обрабатывается микропрограммой, которая выполняется внутри кристалла процессора. Однако если взглянуть на существующие программы через призму статистики, то обнаружится, что из большого набора команд процессоров CISC (у некоторых типов — свыше 300) используется только небольшая, постоянно повторяющаяся часть в размере около 20 процентов.

В дополнение к этому, комплексную программу можно зачастую заменить несколькими эффективными командами, которые в состоянии справиться с поставленной задачей быстрее. Со временем изготовителям модулей памяти удалось резко сократить время доступа, в результате чего в середине 1980-х годов произошел возврат «к истокам», и была разработана архитектура RISC — возникли компьютеры с сокращенным набором команд, у которых команды, как и в процессорах первых поколений, снова декодировались посредством «жесткой прошивки».

Перечислим характерные особенности архитектуры RISC:

- как следует из самого названия, самой примечательной особенностью является ограниченное количество эффективных команд;
- отсутствие классического накапливающего сумматора в пользу большего числа равноправных рабочих регистров;
- организация диапазонов памяти по Гарвардской модели;
- единый интерфейс с запоминающими устройствами за счет исключительного применения команд вида "Загрузка/Сохранение";
- обработка почти всех команд в течение единственного машинного такта;
- оптимизация аппаратного обеспечения и набора команд с целью применения программирования на языках высокого уровня.

Поначалу использование архитектуры RISC было ограничено мощными компьютерами, выступающими в роли рабочих станций, однако вскоре были открыты преимущества этой архитектуры для однокристальных микроконтроллеров, что доказывает семейство AVR компании Atmel.

- Представители семейства AVR обладают ограниченным набором из 131 высокоэффективных команд (для модели ATMega16)
- Благодаря особой архитектуре микропроцессоров AVR, исключен ярко выраженный недостаток обычных процессоров, у которых все арифметические и логические операции выполняются исключительно в так называемом накапливающем сумматоре. Вследствие этого, как правило, после завершения собственно вычислительных операций требуются операции обращения к вспомогательным регистрам или промежуточным запоминающим устройствам. У семейства микроконтроллеров AVR этот недостаток отсутствует, поскольку в вычислениях, кроме накапливающего сумматора, задействованы 32 равноправных рабочих регистра, напрямую связанных с арифметико-логическим устройством (АЛУ).
- Микроконтроллеры AVR работают по Гарвардской архитектуре, что подразумевает разделение памяти для программ и данных. Они используют одноступенчатую конвейерную обработку. Это означает, что во время выполнения команды выполняется загрузка следующей команды из памяти программ (рис. 1.1). Благодаря этому, достигается возможность выполнения команды в течение одного тактового цикла. Первая команда программы выполняется на один тактовый цикл дольше, чем та же команда в другом месте программы, поскольку в этом случае выборка не может быть осуществлена параллельно с выполнением предыдущей команды.

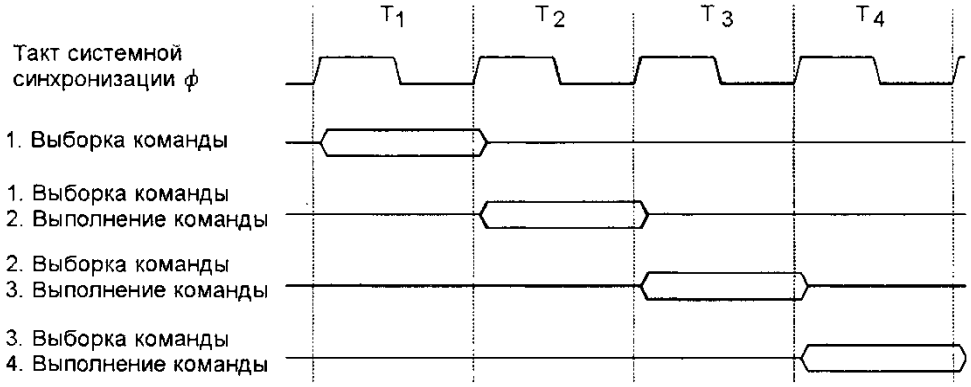


Рис. 1.1 Конвейерная обработка при выборке и выполнении команды в микроконтроллерах AVR

- В микроконтроллерах AVR используются несколько команд загрузки сохранения, с помощью которых к переменным и константам можно обратиться в различных режимах адресации.
- У большинства обычных процессоров и контроллеров такт, сгенерированный кварцевым осциллятором, делится на заранее заданный коэффициент для получения собственно системного такта. Так, например, в контроллере 8051, работающем с частотой кварца 12 МГц, при внутреннем коэффициенте деления минимальное время выполнения команд составляет всего лишь одну секунду, причем "минимальное время" в данном случае подразумевает, что для выполнения некоторого количества команд потребуется больше времени. Однако в случае с микроконтроллерами семейства AVR такого не происходит. В них действительно почти все команды выполняются в течение одного периода частоты кварцевого генератора, что при максимально допустимой частоте 12 МГц в случае микроконтроллера ATMega16 означает всего лишь 83,33 нс. Другими словами, достигается быстродействие до 12 MIPS (Million Instructions Per Second), то есть, — 12 миллионов выполненных команд в течение одной секунды! И, за немногими исключениями, микроконтроллеры семейства AVR действительно обрабатывают все команды в течение единственного системного такта.
- После того как в однокристальных микроконтроллерах для выполнения программ все больше начали использоваться языки высокого уровня (прежде всего, язык программирования С), архитектура AVR была оптимизирована для эффективного взаимодействия аппаратного и программного обеспечения, написанного на языках высокого уровня. К примеру, микроконтроллеры AVR специально для работы с указателями предоставляют в распоряжение программиста команды с функциями автоматического инкремента и декремента.

Структурная схема МК ATMega16

Рассмотрим структуру микроконтроллера, представленную в виде упрощённой схемы на рис. 1.2.

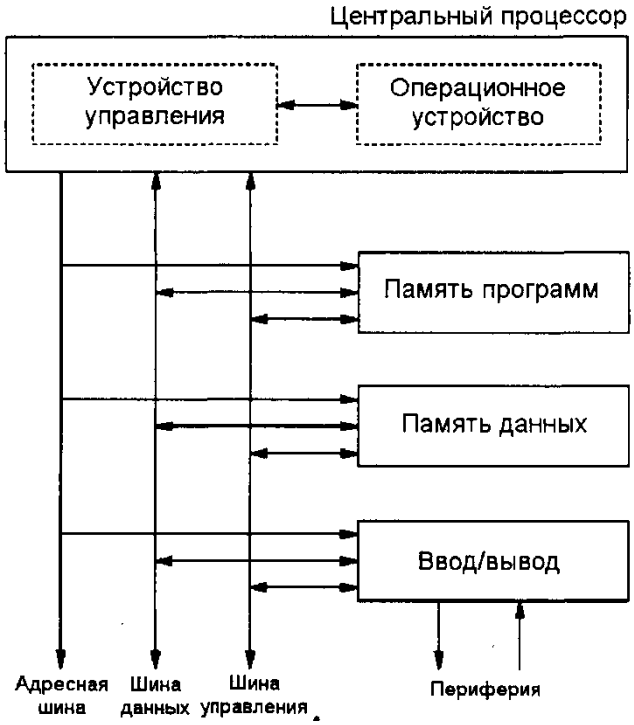


Рис. 1.2 Упрощённая структура микроконтроллера

Система управления, состоящая из счетчика команд и схемы декодирования, берет на себя считывание и декодирование команд из памяти программ, а операционное устройство отвечает за выполнение арифметических и логических операций; интерфейс ввода/вывода позволяет обмениваться данными с периферийными устройствами; и, наконец, необходимо иметь запоминающее устройство для хранения программ и данных.

Более полная структурная схема контроллера представлена на рис. 1.3.

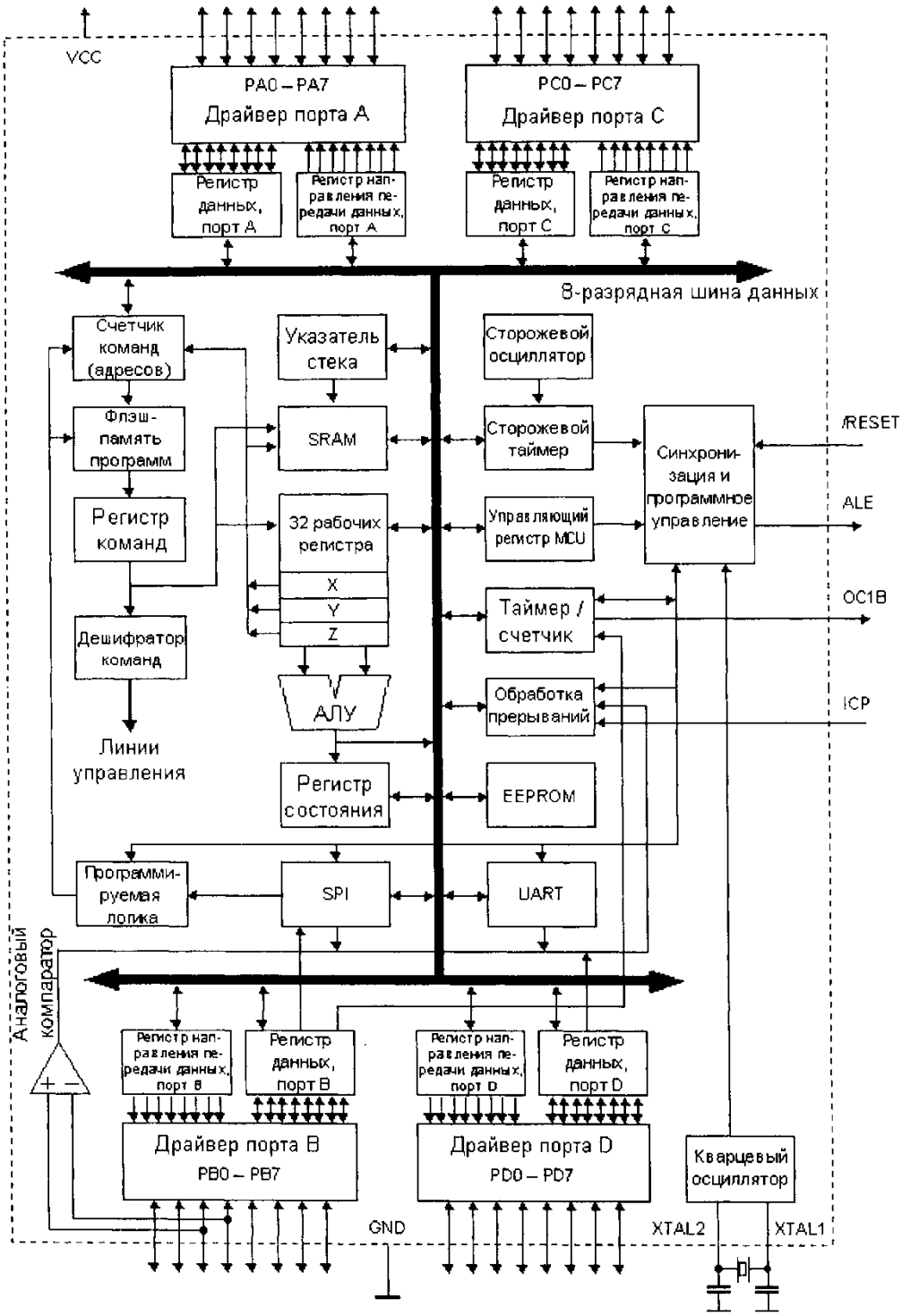


Рис. 1.3 Полная структурная схема МК ATMega16

Рассмотрим более подробно некоторые элементы, из которых состоит микроконтроллер.

Система управления и АЛУ

Система управления регулирует процесс выполнения программы и контролирует взаимодействие отдельных встроенных в кристалл модулей, как это показано на блок-схемах на рис. 1.2 и рис. 1.3. Система координирует выполнение всех действий, необходимых для обработки команды от декодирования до выполнения (например, в случае арифметических операций).

Все арифметические и логические операции выполняются в арифметико-логическом устройстве (АЛУ) и 32-х рабочих регистрах. АЛУ базовой серии микроконтроллеров AVR может выполнять операции сложения, вычитания, смещения, а также логические операции "И", "ИЛИ" и "Исключающее ИЛИ". АЛУ микроконтроллеров AVR является настолько мощным, что в течение единственного системного такта может извлечь из регистров два операнда, выполнить над ними операции и сохранить результат в регистре назначения.

Архитектура памяти

В запоминающих устройствах, соответствующих классической концепции фон Неймана, данные и команды хранятся в одной памяти. В противоположность этому, память по Гарвардской архитектуре, используемой в микроконтроллерах AVR, состоит из нескольких компонентов. В данном случае разделены память команд и память данных, то есть, обращение к командам осуществляется независимо от доступа к данным.

В микроконтроллерах AVR отдельные сегменты памяти устроены по-другому также и физически.

- Память программ реализована на основе программируемой и электрически стираемой флэш-технологии. Во всех микроконтроллерах AVR память 16-разрядная (двухбайтовая), и всегда находится "на кристалле". Расширение памяти программ с помощью блоков EPROM или флэш невозможно.
- Внутренняя память для энергозависимых данных (то есть, данных, которые будут потеряны после отключения рабочего напряжения), представляет собой статическую память RAM (SRAM). Преимущество этого заключается в том, что отпадает всякая необходимость во внутренней регенерации как в случае с некоторыми другими процессорами, которые используют динамическую память. По этой причине микроконтроллеры AVR могут работать с тактами вплоть до 0 Гц. В дополнение к этому, некоторые микроконтроллеры AVR для увеличения объема обрабатываемых данных могут работать с подключаемой внешней памятью SRAM.
- Для данных, которые должны сохраниться после отключения рабочего напряжения, в распоряжении имеется микросхема EEPROM (Electrically EPROM — электрически стираемое ППЗУ). В память EEPROM можно записывать данные во время нормального выполнения программы. Для области EEPROM также нет обязательной необходимости в программирующем устройстве.

Обзор статической памяти (SRAM)

Статическое ОЗУ (SRAM) семейства микроконтроллеров AVR, как это показано на рис. 1.4, представляет собой блок со сквозной адресацией, состоящий из трех подобластей.

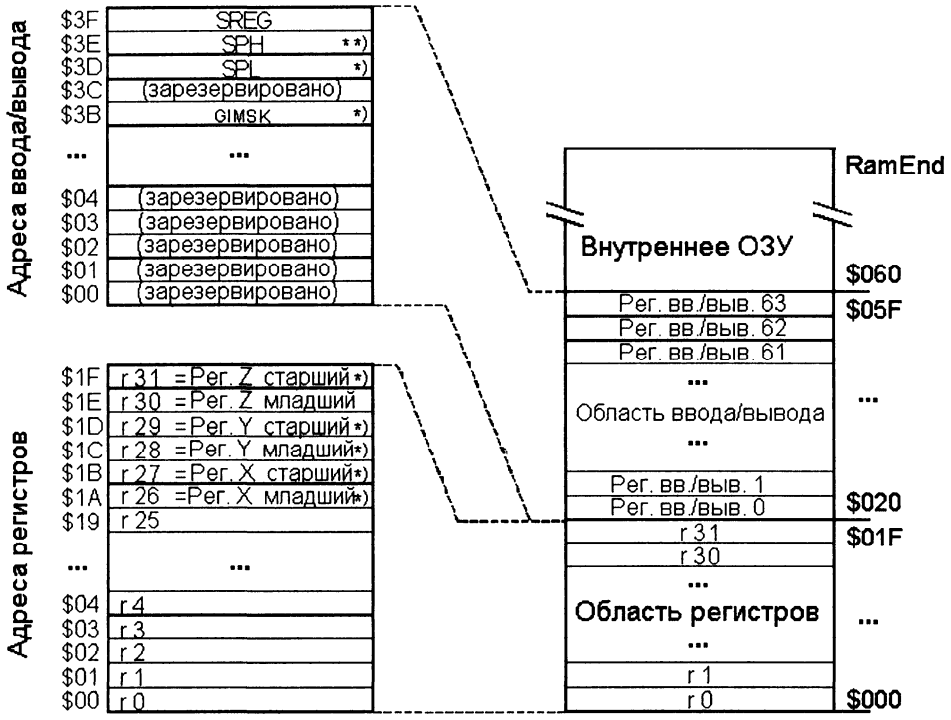


Рис. 1.4 Память SRAM

Первая 31 ячейка памяти адресует файл регистров общего назначения, далее следует блок из 65 ячеек, адресующих память ввода/вывода, а в конце расположен блок из 1024 ячеек, адресующих память данных.

Поддерживается пять типов адресации памяти: прямая, косвенная со смещением, косвенная, косвенная с пред-декриментом и косвенная с пост-инкриментом. Мы не будем подробно останавливаться на режимах адресации, так как при написании программ на языке высокого уровня компилятор сам определяет тип адресации, наиболее подходящей для каждой ситуации.

Регистры общего назначения

Самая нижняя область памяти SRAM образует регистровый файл с 32 рабочими регистрами, которые все связаны с АЛУ и доступ к которым может быть выполнен в течение единственного такта системной синхронизации. Это означает, что за время такта в арифметико-логическое устройство вводятся два операнда из регистрового файла, выполняется операция, а результат запоминается в регистре назначения.

Несмотря на то, что рабочие регистры, с физической точки зрения, не являются ячейками памяти, им, как это показано на рис. 1.4, поставлены в соответствие 32 самых нижних адреса от $00 до $1F в памяти SRAM, поэтому их можно адресовать как обычные ячейки.

Регистры ввода-вывода

В этой области памяти SRAM размещены все регистры для программирования, управления и сигнализации обо всех периферийных функциях микроконтроллеров AVR.

К регистрам ввода/вывода относятся регистры разрешения/запрета отдельных прерываний, а также указатель стека, регистры состояния для указания результатов арифметических и логических операций, регистры управления работой многочисленных компонентов аппаратного обеспечения и периферии (порты, таймер, UART, интерфейс SPI, аналоговый компаратор, сторожевой таймер, режимы пониженного энергопотребления), а также регистры для обращения к памяти EEPROM.

В разделах, описывающих регистры, авторы многих книг по микроконтроллерам стараются вывалить на читателя как можно больше информации. В ход идут огромные таблицы, перечисляющие каждый из доступных регистров, а некоторые после этого ещё и рассматривают каждый регистр в отдельности. Мы не будем заниматься таким неблагодарным делом, так как считаем переписывание документации контроллера пустой тратой времени. Лучший способ изучить какую-то вещь — это начать с ней работать. Поэтому мы будем рассматривать различные регистры по мере того, как в них будет появляться необходимость. А если вам понадобиться справочник по регистрам, то лучшего места, чем документация контроллера, всё равно не найти.

Память данных

Как и следует из названия, в данном блоке расположены динамически изменяющиеся данные, которые используются при выполнении программы. Лучшая аналогия — это оперативная память, которая присутствует в каждом компьютере.

В памяти данных также размещается стек. Стек служит для хранения адресов возврата при вызове подпрограмм, а также для передачи параметров в подпрограммы. Кроме того, зачастую в стеке, как в буфере, временно хранятся переменные и промежуточные результаты. Более подробно о стеке будет рассказано в разделе, посвящённому программированию на С.

Перезаписываемая Flash-память программ

В этой памяти расположен скомпилированный исполняемый код программы. Как правило, эта область памяти не подлежит изменению в нормальном режиме работы контроллера (за исключением случая self-programming). После компиляции программы бинарный файл записывается во flash-память с помощью специального устройства — программатора. Более подробно этот процесс будет рассмотрен далее, в разделе, посвящённому программированию и отладке контроллера.

Энергонезависимая память (EEPROM)

Микроконтроллер ATMega16 содержит 512 байт программируемой энергонезависимой памяти (EEPROM — Electrically Erasable Programmable Read-Only Memory). Она может использоваться для хранения небольших объёмов данных, которые изменяются достаточно редко. В качестве примера можно привести градуировочные таблицы для некоторых датчиков, имеющих нелинейные характеристики.

Периферия микроконтроллера

В этом разделе мы дадим небольшой поверхностный обзор средств, которые представляет разработчику микроконтроллер. Более подробную информацию о периферии можно найти в соответствующих разделах пособия, или в документации контроллера. Итак, вот краткий список средств, с которыми можно работать.

Порты ввода-вывода общего назначения

Микроконтроллеры Mega16 имеют четыре 8-битных порта ввода-вывода. Каждому биту соответствует свой вывод контроллера, который может находиться в трёх состояниях: высокий уровень (Logical High), низкий уровень (Logical low) и неопределённое состояние (Intermediate state). Каждый из выводов может работать в одном из двух режимов: ввод (input) или вывод (output). Кроме того, все выводы имеют альтернативные функции, которые можно использовать, управляя специальными регистрами.

АЦП

Альтернативной функцией порта А является работа в режиме АЦП. Каждый из восьми выводов этого порта может использоваться как независимый канал измерения. Диапазон измеряемого напряжения варьируется от 0 вольт до напряжения питания. Также на двух каналах может применяться встроенный усилитель, позволяющий усилить входной сигнал в 10 или 200 раз.

Программируемые таймеры

Контроллер Mega16 содержит три программируемых таймера-счётчика, которые могут отсчитывать интервалы времени, работать в режиме ШИМ и управлять некоторыми выводами микроконтроллера при наступлении определённых условий. Два таймера имеют разрядность 8 бит, а один — 16 бит. В качестве тактового сигнала для таймеров может быть выбран внешний сигнал, или системный тактовый сигнал с возможностью использования делителя частоты.

Внешние прерывания

Механизм внешних прерываний позволяет прервать нормальный ход выполнения программы и выполнить указанный код при наступлении определённого события. Таким событием обычно является изменение логического уровня на одном из двух выводов микроконтроллера: INT0 или INT1.

USART

Контроллер Mega16 имеет универсальный последовательно-параллельный приёмо-передатчик, с помощью которого можно реализовать коммуникацию по различным протоколам, например RS-232. Данный интерфейс часто применяется для связи с компьютером, терминалом или отладочным устройством.

SPI

Serial Peripheral Interface — это интерфейс, с помощью которого контроллер может осуществлять коммуникацию с другими устройствами. По этому интерфейсу могут работать самые разные устройства: от модулей памяти до программаторов и отладчиков.

Watchdog timer

Это специальный таймер, который может перезагружать микроконтроллер по прошествии определённого времени. Эта возможность позволяет предотвратить зависание контроллера, так как таймер работает независимо от исполняемой программы.

Аналоговый компаратор

Микроконтроллер содержит также аналоговый компаратор, который может сравнивать значения напряжения на положительном входе AIN0 и отрицательном AIN1. Если напряжение на AIN0 больше, чем напряжение на AIN1, компаратор устанавливает бит ACO в регистре ACSR. Кроме того, компаратор может вызывать прерывания по заданному фронту изменения сигнала.

Обзор языка С

Введение

В этой главе мы дадим краткий обзор языка С применительно к программированию микроконтроллеров. За многие годы было написано большое количество книг по этому языку, поэтому нет смысла писать ещё одну, подробно рассматривая каждую мелочь. Вместо этого, в начале каждого раздела будет приводиться небольшой пример, демонстрирующий тот или иной аспект применения С, после чего этот пример будет подробно разобран и объяснён. Таким образом, с первой же страницы мы начнём писать реальные программы для микроконтроллера, не забивая голову теоретическим материалом, который может никогда не пригодиться.

Первая программа

Традиционно при изучении нового языка программирования первой программой становится одна-единственная строка, выводящая на экран сакральное “Hello world!“. Однако у микроконтроллера нет ни экрана, ни вообще какого-либо стандартного средства вывода. Поэтому сейчас мы ограничимся написанием чуть более сложной программы, состоящей из целых 11 строк.

```
int main()
{
	int a = 2;
	int b = 3;
	int c = 1;
	ind D;

	D = (b * b) - (4 * a * c);

	retun 0;
}
```

Эта небольшая программа находит дискриминант квадратного уравнения при заданных величинах *a*, *b* и *c*. Конечно, она не несёт никакой практической ценности, так как значения «зашиты» в программу на этапе компиляции, однако, в качестве примера эта программа позволит нам понять некоторые ключевые концепции языка С.

Разберём этот код построчно. На первой строке мы видим объявление **функции**. Функция — это основной «строительный блок», из которых собирается программа. Она представляет собой отдельный участок кода, к которому можно обратиться по имени из другого места программы. Перед именем функции *main* расположен **тип возвращаемого ей значения**. На данный момент не будем заострять на этом внимание, и вернёмся к этому вопросу позже, при более подробном рассмотрении функций. Скажем лишь, что любая функция имеет вид:

```
Тип	имя_функции([список_параметров])
{
	Код_функции
}
```

Код, который непосредственно выполняется функцией, заключается в фигурные скобки *{ }*. Эти скобки используются для выделения любого логического блока в программе, и мы ещё вернёмся ним при рассмотрении других примеров.

В самом начале блока, обозначенного фигурными скобками, расположены **объявления переменных**. Переменная — это просто участок памяти, к которому мы можем обратиться по имени. Размер этого участка определяется **типом** переменной. Например, если мы объявим переменную *int foo*, в памяти будет выделен блок размером два байта — именно столько занимает тип данных *int*. Теперь, если мы будем обращаться в коде к переменной *foo*, компилятор поймёт, что ему необходимо обратиться к определённой области памяти, выделенной именно под эту переменную.

Таким образом, тип переменной отвечает за то, сколько памяти будет выделено на хранение информации, содержащейся в этой переменной. Стандарт С определяет несколько основных типов данных.

Тип данных	**Размер выделяемой памяти**	**Описание**
char	1 байт	Обычно используется для хранения символьных значений
int	2 байта	Целочисленный тип
float	4 байта	Число с плавающей точкой
double	8 байт	Число с плавающей точкой двойной точности

Существует также некая абстрактная величина, называемая *словом*. Размер слова исчисляется в битах или байтах и определяет разрядность основных регистров процессора, максимальный объём адресуемой памяти и ещё много интересных вещей. Тип данных *int* обычно равен размеру слова на конкретной платформе. В случае 8-битных контроллеров ATMega слово занимает два байта, столько же занимает и *int*. Все эти сложные слова были сказаны для того, чтобы подчеркнуть тот факт, что тип *int* является «родным» для контроллера, и именно операции над переменными этого типа производятся быстрее всего.

Стоит заметить, что при программировании микроконтроллеров типы данных с плавающей точкой используются довольно редко, так как операции с ними отнимают слишком много ресурсов. Для работы с дробными значениями чаще применяют числа с фиксированной точкой, реализуемые при помощи переменных типа *int* и побитового сдвига.

Четыре базовых типа можно расширить с помощью *спецификаторов*. Спецификатор — это ключевое слово, которое располагается перед базовым типом переменной. Существует 4 спецификатора: *signed, unsigned, short, long*. Первые два влияют на диапазон значений, принимаемых переменной, а два других — на размер памяти, выделяемой для переменной.

Количество байт, выделяемых для переменной, влияет на диапазон значений, которые эта переменная может принять. Например, для типа *char* выделяется один байт памяти. Как известно, один байт состоит из восьми бит, а каждый бит, в свою очередь, может принимать два значения: 1 или 0. Таким образом, набор из восьми бит может принять 2^8, то есть 256 возможных значений. Значит, если мы хотим хранить в

переменной типа *char* как отрицательные, так и положительные числа, в один байт у нас влезет диапазон значений от -128 до 127. Если же мы хотим хранить только положительные значения, у нас будет диапазон от 0 до 255. Как вы уже догадались, мы вплотную подошли к пониманию смысла спецификаторов *signed* и *unsigned*. Спецификатор *signed* указывает на то, что в переменной могут храниться как положительные, так и отрицательные значения, тогда как *unsigned* разрешает переменной принимать лишь положительные значения.

Особо любознательные найдут интересным тот факт, что в памяти значения signed и unsigned переменных одного типа имеют один и тот же вид. Например, содержимое байта памяти:

```
Номер бита →      7 6 5 4 3 2 1 0
Значение бита →   1 0 0 1 1 0 1 1
```

будет интерпретировано как число -101, если эта область памяти принадлежит переменной типа *signed char*, и как число 155 в случае *unsigned char*. По сути, спецификаторы *signed* и *unsigned* определяют интерпретацию последнего, восьмого бита. В случае signed-переменной «1» в этом бите означает отрицательное число, а «0» — положительное. Младшие 7 бит определяют значение. В случае же unsigned-переменной значение восьмого бита принимает полноправное участие в формировании значения байта.

Наверное, это место будет наиболее удачным для того, чтобы напомнить, как формируется десятичное значение из набора нулей и единиц. У каждой цифры («0» или «1») в двоичном числе есть определённый *вес*, который представляет собой возрастающие степени числа 2 и увеличивается с номером цифры (бита). Биты нумеруются справа налево.

Номер бита, n	→	7	6	5	4	3	2	1	0
Вес бита, 2^n →		128	64	32	16	8	4	2	1
Значение бита	→	1	0	0	1	1	0	1	1

Чтобы получить десятичное значение, нужно сложить веса всех битов, имеющих значение «1». Найдём значение числа из примера, предположив, что оно беззнаковое. Как видно, в единицу выставлены биты 0, 1, 3, 4 и 7. Складываем их веса: 1 + 2 + 8 + 16 + 128 = 155. Возможно, это не самое распространённое описание систем счисления, однако, как показывает практика, оно более понятно, чем формальные тексты учебников.

Вернёмся теперь к спецификаторам типов. Мы уже выяснили, как работают спецификаторы *signed* и *unsigned*, разберёмся теперь с *long* и *short*. Эти два спецификатора влияют на размер памяти, выделяемой для переменной. Например, для *short int* будет выделено 2 байта, в то время как для *long int* — 4. Эти два спецификатора неприменимы к типу *char*. Для наглядности приведём табличку.

Тип данных	Размер памяти	Диапазон значений
signed char	1 байт	-127 – 127
unsigned char	1 байт	0 – 256
signed short int	2 байта	-32767 – 32767
unsigned short int	2 байта	0 – 65535
signed long int	4 байта	-2147483647 – 2147483647
unsigned long int	4 байта	0 – 4294967295

В данном курсе мы не будем использовать типы с плавающей точкой, поэтому они не вошли в таблицу.

По умолчанию переменные типа *char* имеют спецификатор *signed*, а переменные типа *int* — *signed* и *short*. Таким образом, эти записи будут эквивалентны:

```
signed int foo;     // explicitly signed and short by default
int bar;            // signed by default and short by default
short baz;          // signed by default and explicitly short
```

Ещё один важный аспект, касающийся переменных, и требующий внимания — это их инициализация. В простейшем случае, переменную можно объявить двумя способами:

```
int foo;
int bar = 123;
```

В первом случае происходит только выделение памяти для переменной *foo*, однако в эту память не записывается никакого значения. Это означает, что наша переменная будет содержать то значение, которое хранилось в этой области памяти до её объявления. При написании программы мы ни в коем случае не должны делать предположений о значениях неинициализированных переменных. При запуске программы переменная *foo*, скорее всего, будет содержать значение *0xFFFF*. Если переменная объявлена внутри функции, то есть вероятность, что при последующих вызовах для неё выделится та же область памяти, и тогда «новая» переменная будет иметь значение своей прошлой «инкарнации». Эта особенность отличает C от некоторых других языков, которые автоматически присваивают объявляемым переменным значение по умолчанию (например, 0 для численных типов).

Мы также можем явно присвоить значение переменной при её объявлении. Для этого нужно просто поставить знак равенства после её имени и написать нужное значение. Следующие две записи будут эквивалентны:

```
int bar = 123;

int bar;
bar = 123;
```

Последний вопрос, который мы должны обсудить перед получением зелёного пояса по переменным — это их область видимости. Рассмотрим ещё один пример.

```
int foo = 123;

int main() {
	int bar = 456;
	int result = foo + bar;
	int mainstuff = 911;

	return 0;
}

int notmain() {
	int bar = 789;
	int result = foo + bar;

	result = result + mainstuff; // Whooopsie.

	return 0;
}
```

Переменные могут быть двух типов в зависимости от их области видимости: **глобальные** и **локальные** (на самом деле, всё несколько сложнее, но пока этим не стоит забивать голову).

Глобальные переменные можно использовать в любой функции в пределах одного файла. Глобальные переменные объявляются в самом начале файла, перед всеми функциями. В примере, приведённом выше, *foo* — глобальная переменная.

Локальные переменные объявляются внутри блока, ограниченного фигурными скобками, и могут использоваться только внутри этого блока. В нашем примере внутри функции *main* объявлена переменная *bar* со значением 456. В переменную *result* записывается результат сложения глобальной переменной *foo* и локальной переменной *bar*. Для функции *main* значение *result* будет равно 579.

Функция *notmain* очень похожа на функцию *main*: она тоже объявляет переменную *bar*, и тоже складывает её значение с глобальной переменной. Однако, в отличие от *main*, значение переменной *bar* в *notmain* равно 789, а значит результат будет равен 912. Как видите, обе функции используют одно значение глобальной переменной *foo* и разные значения собственных локальных переменных *bar*.

Студент, писавший функцию *notmain*, прогуливал лекции по С, поэтому решил использовать в ней переменную *mainstuff*, локально объявленную в *main*. Логично предположить, что его программа не скомпилируется, и он будет ходить на пересдачу до тех пор, пока более опытные товарищи не расскажут ему про область видимости переменных.

На этой оптимистичной ноте мы и закончим обзор переменных, а в следующем разделе рассмотрим операции, которые с ними можно производить.

Операции над переменными

Над переменными можно проводить как тривиальные арифметические операции (как в первом примере прошлого раздела), так и более сложные, логические. Однако прежде, чем мы начнём говорить об операциях, стоит обсудить такое, на первый взгляд, простое действие как присваивание.

Очевидно, что переменной можно присвоить значение, написав после неё знак «равно» и подставив нужное значение. Но что будет, если написать следующий код?

```
unsigned int foo = 65535;
unsigned char bar = foo;
```

Мы объявили переменную *foo* типа *int*, присвоили ей значение, а потом присвоили значение переменной *foo* переменной *bar*, имеющей тип *char*. Но разве так можно делать? Ведь переменная типа *int* занимает два байта, тогда как переменная типа *char* — всего один! Оказывается, так делать можно, и есть одно-единственное правило, которое действует во всех подобных случаях. Оно заключается в том, что присваиваемое значение приводится к типу переменной, стоящей слева от знака «равно». В нашем случае от значения *foo* будет взято только 8 младших бит и они будут присвоены *bar*. Таким образом, переменная *bar* после присваивания будет иметь значение 255.

```
Номер бита →      15  14  13  12  11  10  9   8     7  6  5  4  3  2  1  0
Значение бита →   1   1   1   1   1   1   1   1     1  1  1  1  1  1  1  1
                   -------------------------------    -----------------------
                     Эта часть отбрасывается          А эта присваивается bar
```

Ещё один вопрос, который стоит обсудить, пока мы говорим о присваивании — это форма записи числовых констант. При использовании компилятора AVR GCC можно записывать числа тремя способами: в десятичной системе (как мы до сих пор делали во всех примерах), в двоичной и шестнадцатеричной системах.

Запись чисел в двоичной системе особенно удобна при программировании микроконтроллеров, где часто приходится задавать значения отдельных битов в регистрах, отвечающих за те или иные настройки. Чтобы записать число в двоичной системе, перед ним нужно поставить префикс *0b*.

```
unsigned char foo = 123;
unsigned char bar = 0b01111011;
unsigned char baz = 0x7b;
```

В этом примере все три переменные имеют одинаковые значения. При присвоении значения переменной *bar* мы использовали двоичную форму записи, которая позволила нам наглядно установить значение каждого бита. Для переменной *baz* мы использовали шестнадцатеричную форму записи с префиксом *0x*. Прочитать больше про шестнадцатеричную систему счисления можно в Википедии.

Арифметические операции

Теперь приведём сводную таблицу простейших арифметических операторов.

Оператор	**Описание**
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
++	Инкремент
--	Декремент

Большинство операций изучались ещё в первом классе, но на некоторых всё же стоит остановиться подробнее.

Например, нужно помнить, что использование целочисленных переменных (таких, как *int*) для хранения результатов математических операций приведёт к отбрасыванию дробной части при делении. В следующем примере переменная *foo* будет иметь значение 3, а не 3,3333(3).

```
int foo = 10.0 / 3;
```

Чтобы получить остаток от деления числа x на число y, можно использовать оператор %. В этом примере значение переменной *foo* будет равно 1.

```
inf foo = 10 % 3;
```

Две операции, присущие большей частью языкам программирования — это инкремент и декремент. Эти операции позволяют увеличить или уменьшить значение переменной на единицу. Следующие записи будут эквивалентны:

```
foo = foo – 1;      // substracts 1 from foo
foo--;       // does exactly the same
```

С инкрементом и декрементом также связан один нюанс, который часто приводит к трудноуловимым багам в программах. Дело в том, что эти два оператора могут располагаться как перед именем переменной, так и после него. Такие формы записи называются префиксной и постфиксной. Они не имеют значения, если вы просто увеличиваете или уменьшаете значение переменной. Однако форма оператора инкремента или декремента имеет значение, если вы сразу присваиваете результат операции другой переменной.

```
int foo = 13;
int bar = foo++;
int baz = ++foo;
```

На второй строке мы сначала присваиваем *bar* текущее значение *foo*, и лишь затем увеличиваем его на единицу. Таким образом, после выполнения второй строки *bar* будет иметь значение 13, а *foo* — 14. На третьей строке мы используем префиксную форму записи, а значит, сначала увеличиваем значение *foo*, и только после этого присваиваем его *baz*. После выполнения третьей строки обе переменные, *baz* и *foo*, будут иметь значение 15.

Вопрос на зачёт: какое значение примет bar?

```
int foo = 5;
int bar = ++foo + ++foo;
```

При применении арифметических операций нужно также помнить о переполнении. Например, после выполнения этого кода переменная *foo* будет иметь значение 0, так как верхняя граница для типа *unsigned int* равна 65535.

```
unsigned int foo = 65535;
foo++;
```

Кроме обычных арифметических операторов в языке C существует набор сокращённых операторов. Они используются в тех случаях, когда нужно произвести какое-то действие с текущим значением переменной и сохранить результат в эту же переменную. Например, следующие две записи будут эквивалентны.

```
foo = foo + 5;      // Add 5 to foo
foo += 5;           // This adds 5 to foo as well
```

Любой оператор, который можно применить к двум операндам, можно также записать и в сокращённой форме.

```
foo += 5;
bar -= 10;
baz /= 2;
foo *= 3;
bar &= 1;
baz |= 1;
```

Операции сравнения

В языке C есть несколько классов операций, результатом которых становится булевское значение (истина/ложь*; true/false). Стандарт C89 определяет значение *false* как ноль, и значение *true* как любое другое число.

```
int foo = 0;        // equals FALSE
int bar = 1; // equals TRUE
int baz = -32;      // equals TRUE as well
```

Операциями, возвращающими значения *true* и *false*, являются, например, операции сравнения.

Оператор	**Описание**
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
!=	Не равно

Смысл большей части перечисленных операторов очевиден, особое внимание стоит уделить лишь оператору «равно». Всегда нужно помнить, что для присвоения значения переменной используется «одинарное равно», а для сравнения двух переменных — «двойное». Дело усугубляется тем, что компилятор не выдаст ошибки при неправильном использовании оператора сравнения.

```
int foo = 0;

if (foo = 5) {
      // do the course project
} else {
      // have a beer
}
```

Приведённый код скомпилируется, однако, будет работать совсем не так, как планировал автор. Вместо того чтобы сравнить значение *foo* (которое изначально равно 0, то есть *false*) с числом 5, компилятор сначала присвоит *foo* значение 5, а потом обработает инструкцию *if*. Так как 5 не равно 0, всё выражение будет трактовано как *true*. Правильный код будет выглядеть так:

```
int foo = 0;

if (foo == 5) {
      // do the course project
} else {
      // have a beer
}
```

В остальном операторы сравнения работают очень просто: выдают *true*, если выражение соответствует истине, и *false* — если нет.

Логические операции

Второй класс операций, которые возвращают булевские значения — это логические операции. Существует три основные логические операции, поддерживаемые языком C.

Оператор	**Описание**
&&	Логическое «И» (AND)
||	Логическое «ИЛИ» (OR)
!	Логическое «НЕ» (NOT, отрицание)

Каждый из этих операторов может применяться к булевским значениям (а учитывая «лояльное» определение стандартом булевских значений, вообще ко всем переменным). Оператор *&&* возвращает *true* только в том случае, если оба операнда равны *true*. Оператор *||* возвращает *true* если хотя бы один из операндов имеет значение *true*. Оператор *!* просто возвращает противоположное значение операнда.

Приведём стандартную таблицу истинности.

foo	**bar**	**foo && bar**	**foo || bar**	**!foo**
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

С помощью логических операторов можно делать сложные проверки, например:

```
if (!foo && (bar || baz))
      DestroyTheWorld();
```

В данном случае также иллюстрируется обработка скобок в C. Если в некотором выражении присутствуют скобки, компилятор обрабатывает их на основе правил обычной математики. Поэтому, если вы хотите написать сложное условие, и не знаете как, просто воспользуйтесь здравым смыслом и знаниями, полученными в школе — этого будет вполне достаточно.

Побитовые операции

Мы подошли к изучению самого хитрого класса операторов — побитовых. Эти операторы манипулируют не с целыми значениями переменных, а с их элементарными частями — битами. Понимание этих операций очень важно при программировании микроконтроллеров, поэтому рассмотрим их подробнее.

Побитовые операции очень похожи на логические, однако, они возвращают не булевские значения, а значение переменной после проведения над ней операции.

Оператор	**Описание**
&	Побитовое «И» (AND)
|	Побитовое «ИЛИ» (OR)
~	Побитовое «НЕ» (NOT)
^	Исключающее «ИЛИ» (XOR)
<<	Побитовый сдвиг влево
>>	Побитовый сдвиг вправо

Побитовые операции работают следующим образом: берутся два операнда, после чего к **каждому** из битов применяется указанная операция. Рассмотрим пример.

```
      11010010
&     10110111
      --------
      10010010
```

В этом примере мы взяли два двоичных числа и применили к ним побитовое «И». Для каждой пары бит в этих числах мы применили логическую операцию «И», как если бы эти биты были булевскими переменными. Из результирующих битов (значений *true* и *false*, получившихся после каждой логической операции) и складывается ответ. Приведём ещё один пример.

```
      11010010
|     10110111
      --------
      11110111
```

А вот пример использования побитовой операции в коде программы:

```
unsigned char foo = 0b01110111;
if (foo & 0b01000000)
{
       // do stuff
}
```

Этот пример иллюстрирует очень распространённую технику проверки значения определённого бита, называемую *маскированием*. Допустим, нам нужно узнать, имеет ли бит номер 6 значение «1», и предпринять какие-то действия, если это так. Мы применяем побитовую операцию «И» к интересующей нас переменной и константе, называемой «маской». В маске в «1» выставлены те биты, значение которых нас интересует в переменной *foo* (обычно в «1» выставляется только один бит маски, иначе результат проверки будет неоднозначным). В нашем случае операция будет иметь такой результат:

```
foo          →     01110111
& маска      →     01000000
                   --------
результат    →     01000000
```

Как видите, в результате мы получим десятичное значение 128, которое превратится в *true* при проверке. Рассмотрим случай, когда переменная *foo* будет иметь другое значение:

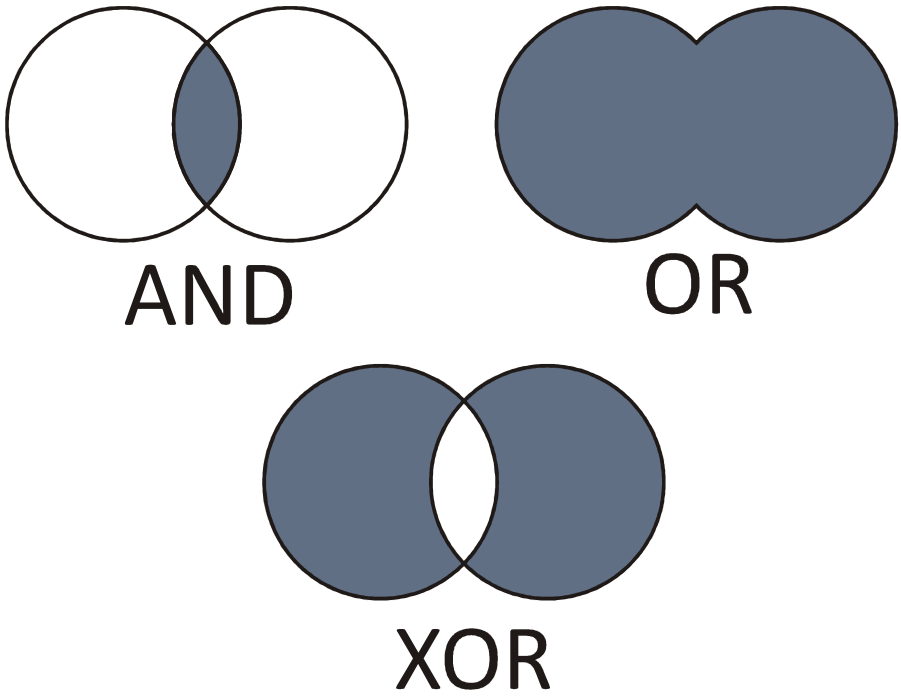
```
foo          →     10110111
& маска      →     01000000
                   --------
результат    →     00000000
```

В этом случае бит, проверяемый маской, равен «0», а значит и весь результат будет равен 0, что эквивалентно *false*.

Таблицы истинности логических операторов OR, AND и NOT соответствуют таблицам истинности соответствующих побитовых операций. Также существует одна хитрая побитовая операция XOR, таблица истинности которой приведена ниже:

a	**b**	**a ^ b**
0	0	0
0	1	1
1	0	1
1	1	0

Для наглядности приведём графическое представление некоторых операций.



Операции сдвига

Осталось рассказать лишь про побитовый сдвиг. Это очень простая, и в то же время мощная операция, суть которой заключается в том, что все биты переменной сдвигаются в ту или иную сторону. Лучше всего эта операция объясняется на примерах.

```
      11001001                                11001001
<< 1                                  >> 1
      10010010                                01100100
<< 2                                  >> 2
      00100100                                00110010
<< 3                                  >> 3
      01001000                                00011001
<< 4                                  >> 4
      10010000                                00001100
```

Как видно из примера, левым операндом является некое значение, а правым — количество бит, на которое нужно произвести сдвиг. Биты сдвигаются в указанную сторону на указанное количество позиций, при этом, биты, вышедшие «за границу», затираются, а на «освободившиеся» места подставляются нули. Побитовый сдвиг имеет также и «физический смысл»: сдвиг влево эквивалентен умножению на 2, а сдвиг вправо — делению на 2 с округлением.

Массивы и указатели

В этом разделе мы рассмотрим одну из самых сложных тем в С — указатели. Есть только одна вещь, которую сделать сложнее, чем понять указатели — это объяснить указатели.

Массивы

Для разминки начнём с более простой темы и разберёмся, что же собой представляют массивы. Массив — это набор переменных одного типа, объединённых под одним именем. Допустим, что каждые пять секунд нам нужно получать некое значение (например, с АЦП) и сохранять его в соответствующую переменную. Решением «в лоб» будет объявить пять переменных и работать с ними:

```
int val1, val2, val3, val4, val5;

if (curSecond == 0)
      val1 = GetADC();
else if (curSecond == 1)
      val2 = GetADC();
// And so on and so far
```

Будем считать, что в переменной *curSecond* за нас кто-то обновляет значение текущей секунды, а функция *GetADC()* получает данные с АЦП. Как видите, здесь мы записываем по одному условию для каждого значения текущей секунды: 0, 1, 2 и т. д.

Однако это не самое удачное решение. Гораздо проще было бы объявить одну большую переменную, которая состояла бы из пяти маленьких частей, верно? Вот именно для этого и существуют массивы. Например, предыдущий пример можно сократить всего до двух строк.

```
int values[5];

values[curSecond] = GetADC();
```

Разберёмся, что же здесь произошло. Сначала мы объявили массив *values*, состоящий из пяти элементов типа *int*. В общем случае массивы объявляются следующим образом:

```
тип имя_массива[количество_элементов];
```

На второй строке происходит присваивание значения АЦП элементу массива, номер которого равен текущему значению *curSecond*. Рассмотрим ещё один пример.

```
int foo[3];
int bar[3] = {123, 456, 789};
int baz;

foo[0] = 123;
foo[1] = 456;
foo[2] = 789;

baz = foo[0];       // baz == 123
baz = bar[1];       // baz == 456
baz = bar[3];       // Index out of bounds, access violation,
                    // nuclear holocaust and bad karma.
```

На первой строке мы объявили массив *foo* из трёх элементов типа *int* и не инициализировали его. Так же, как и в случае неинициализированных переменных, значения элементов неинициализированного массива являются случайными.

На второй строке мы объявили такой же массив, и сразу инициализировали его. Как видите, инициализация делается с помощью фигурных скобок, в которых через запятую перечисляются значения элементов массива.

Далее идёт блок, в котором мы по очереди присваиваем значения элементам массива *foo*. В этом месте нужно сделать одно очень важное замечание: **индексация массивов в C производится с нуля, и никак иначе!** Никаких паскалеподобных компромиссов с явным указанием основания для индексации, которые только запутывают нормальных людей. Итак, ещё раз.

1. При объявлении массива в квадратных скобках указывается **количество** элементов. Если хотите объявить массив из трёх элементов, в квадратных скобках будет значение 3.
2. Доступ к элементам массива в программе производится по номерам, начинающимся с **нуля**. Если вы объявили массив из трёх элементов, то номером последнего его элемента будет **2**.

Как вы уже догадались, последняя строка неверна, так наш массив состоит из трёх элементов, а мы обратились к четвёртому. Нужно очень внимательно следить за индексацией массивов, так как компилятор не производит никакой проверки на выход индекса за границы. Приведенный выше код скомпилируется без единого предупреждения, однако, при попытке выполнить последнюю инструкцию произойдёт нарушение доступа к памяти. Вы не захотите вылавливать подобные баги в прошивке микроконтроллера, уж поверьте.

Как уже было показано в одном из примеров, для доступа к элементам массива можно использовать не только явно заданные значения, но и значения других переменных (в 90% случаев массивы используются именно по такому сценарию).

```
int foo[5] = {1, 2, 3, 4, 5};
int bar = 2;
int baz;

baz = foo[2];
baz = foo[bar];
```

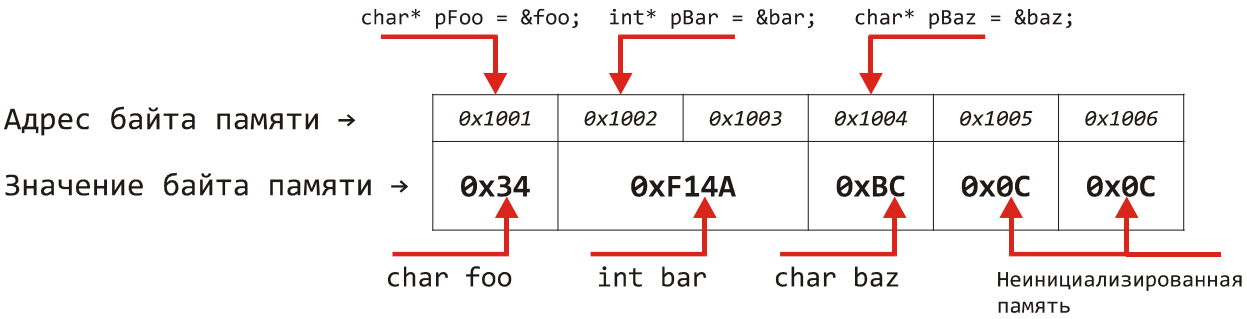
В этом примере мы объявили массив и присвоили переменной *baz* значение одного из его элементов двумя способами. Очевидно, что последние две строки дают одинаковый результат.

Указатели

В самом первом разделе, посвящённом переменным, мы рассказали вам не всю правду. Мы сказали, что при объявлении переменных для них выделяются области памяти, однако, умолчали о том, как к ним происходит доступ на самом деле. Пришло время исправить эту ситуацию.

В памяти нет никакой информации об именах переменных, равно как нет её и в скомпилированной программе. Машинные инструкции ничего не знают об именах, которые вы ввели для своего удобства. Эти инструкции обращаются к ячейкам памяти по их *адресам*. В реальной жизни, чтобы куда-нибудь попасть, вам нужно знать адрес места, в которое вы хотите попасть. Так же происходит и с машинными инструкциями. Чтобы получить или изменить значение переменной, они используют их адреса.

Каждая ячейка памяти (читай: каждый байт) имеет свой уникальный *адрес*, по которому к этой ячейке можно получить доступ. Адрес — это самое обычное число, которое обычно записывается в шестнадцатеричной форме.



На этом рисунке схематично представлен блок памяти, разделённый на элементарные ячейки (байты). Когда мы объявляем переменные обычным способом, происходит выделение необходимой памяти и переменные содержат значения, хранящиеся в выделенных ячейках. На самом деле, обращение к этим ячейкам происходит по их адресам, но эти подробности от нас скрыты.

Однако бывают случаи, когда возникает необходимость работать непосредственно с адресами ячеек памяти. Именно для этого и служат указатели. *Указатель* — это переменная, которая содержит не значение ячейки памяти, а её *адрес*. Рассмотрим пример.

```
int  foo = 5;
int* bar = &foo;

foo = 6;      // *bar == 6, because
              // bar holds foo's address

*bar = 7;     // foo == 7 now, but bar remains the same
              // because foo's address didn't change
```

На первой строке мы объявили обычную переменную *foo* и присвоили ей значение 5. На второй строке мы объявили указатель на переменную типа *int* и присвоили ему адрес переменной *foo*.

Как видите, переменная-указатель объявляется как обычная переменная с добавлением звездочки (*) после типа переменной. Здесь стоит обратить внимание на

небольшой нюанс. Дело в том, что C очень лояльно относится к пробелам, символам табуляции и перевода строки. Поэтому указатели можно объявлять по-разному:

```
int* foo;          // foo is a pointer
int *bar;          // bar is also a pointer
int * baz;         // Even baz is a pointer
int* a, b, c;      // Only a is a pointer here!
```

Звёздочку можно ставить как возле типа переменной, так и возле имени переменной. Использование конкретного стиля — личное дело каждого программиста. Однако нужно помнить, что при объявлении нескольких указателей на одной строке (строка 4 в примере) звёздочку надо ставить возле имени **каждой** переменной, которую вы хотите сделать указателем. В приведённом примере указателем будет только переменная *a*, в то время как *b* и *c* будут иметь обычный тип *int*. Из-за такой путаницы указатели рекомендуется объявлять отдельно, по одному на строке.

Вернёмся теперь к первому примеру. Мы уже выяснили, что переменная *bar* является указателем типа *int*, а значит, может содержать адрес любой переменной типа *int*. На самом деле, в указателях любого типа хранятся адреса одинакового размера, тип указателя играет роль лишь в интерпретации значения, получаемого по этому адресу. Например, в указателе типа *char** хранится адрес байта памяти. Когда мы захотим получить значение ячейки, на которую указывает этот адрес, компилятор автоматически сгенерирует код для получения ровно одного байта по указанному адресу (так как тип *char* занимает один байт). Если же у нас есть указатель типа *int**, в нём будет храниться всё тот же адрес одного байта памяти, однако, при получении значения по этому адресу будут считаны два байта (так как тип *int* занимает 2 байта).

Теперь рассмотрим подробнее способы работы с указателями. Существуют два основных действия: *взятие адреса переменной* и *разыменовывание указателя*. Первое позволяет получить адрес ячейки памяти, в котором хранится значение переменной, а второе — изменить значение этой ячейки при известном адресе. В нашем примере при объявлении указателя *bar* мы сразу инициализировали его адресом переменной *foo*. Как вы уже догадались, для получения адреса переменной используется оператор “&”.

```
тип_указателя* имя_указателя = &имя_переменной;
```

Оператор “&” можно поставить перед именем любой переменной, и результатом этого действия будет адрес первой из ячеек памяти, которые занимает данная переменная.

Для доступа к значению памяти, адресуемой указателем, существует оператор разыменовывания “*”. Для того, чтобы получить или изменить значение памяти, на которую ссылается указатель, достаточно поставить звёздочку перед именем этого указателя.

```
*имя_указателя = значение_переменной;
```

Символ “*” довольно популярен в C: он используется для объявления указателя, и он же применяется для его разыменовывания.

Имея такой багаж знаний, ещё раз посмотрим на первый пример с указателями. Очень важно понимать, что после того, как мы присвоили указателю *bar* адрес переменной *foo*, *bar* содержит именно **указатель** на значение, а не само значение. Таким

образом, если мы изменим значение переменной *foo*, указатель *bar* не изменится, так как переменная *foo* осталась на своём месте в памяти, мы просто изменили *значение* этой памяти. Если же мы разыменуем указатель *bar* с помощью оператора “*”, то мы, по сути, будем присваивать значение переменной *foo*, так как именно на него указывает *bar*. Именно поэтому после выполнения инструкции “*bar = 7;” переменная *foo* тоже примет значение 7.

Связь между указателями и массивами

Мы не случайно объединили темы указателей и массивов в один раздел. Между ними существует тесная связь, о которой мы сейчас и поговорим. До сих пор использование нами массивов сводилось к доступу к их элементам по индексу.

```
int foo[3];

foo[0] = 1;
foo[1] = 2;
foo[2] = 3;
```

Но что представляет собой сама переменная *foo*, если записать её отдельно, без квадратных скобок? А представляет она собой не что иное, как указатель, в котором хранится адрес первого элемента массива. Зная эту тайну, мы можем написать следующий код.

```
int foo[3] = {1, 2 ,3};
int* bar;

bar = foo;          // No “&” operator here!
bar[0] = 123;       // Now we can use bar like it was
                    // an array from the start!
```

Здесь мы объявили массив *foo* типа *int* и указатель *bar* того же типа. Зная, что *foo* — это такой же указатель, мы можем спокойно присвоить его значение указателю *bar*. После чего мы можем использовать указатель *bar* с квадратными скобками для доступа к элементам массива *foo*. Да, всё верно, на низком уровне переменные *foo* и *bar* представляют собой одно и то же за тем лишь исключением, что указателю *bar* вы можете присвоить адрес любой переменной, а *foo* всегда будет указывать на первый элемент своего массива. Рассмотрим ещё один комплексный пример.

```
int foo[3] = {1, 2, 3};
int* bar;
int baz;

bar = foo;          // can me haz array?

*bar = 123;         // We changed the first element of the array
                    // foo[0] == 123 now

bar[1] = 456;       // foo[1] == 456

*(bar + 2) = 789;   // Hardcore pointer arithmetic,
                    // foo[2] == 789

baz = bar[2];       // This one is simple, baz == bar[2] == foo[2] == 789;

bar = &foo[1];      // bar now points to the second element of foo

bar[1] = 0;         // foo[3] == 0. Yes, that’s right.
                    // If you understood this, you’ll probably
                    // make it till the end of the semester ;)
```

Некоторые моменты здесь требуют объяснения. Например, в инструкции

```
*(bar + 2) = 789;
```

мы использовали так называемую указательную арифметику. Кроме обычного присваивания указателей с ними можно проводить операции сложения, вычитания, инкремента и декремента (и никаких других). При проведении этих операций адрес, хранящийся в указателе, изменяется не на то количество байт, которое указано после знака «+» или «-», а на размер, занимаемый типом данных, умноженный на число после знака. Например, если у нас есть указатель типа *int* foo*, в котором хранится адрес 0x1001, то после выполнения инструкции “foo + 2“ *foo* будет хранить не 0x1003, а 0x1005, так как тип *int*, с которым объявлен указатель, занимает два байта. Таким образом, в нашем примере мы прибавили к указателю первого элемента значение 2, перейдя, таким образом, к третьему элементу, а затем разыменовали указатель, присвоив значение 789 третьему элементу массива.

Инструкция

```
bar = &foo[1];
```

демонстрирует нам, что квадратные скобки, по сути, действуют как оператор разыменования. Здесь мы сначала получили значение первого элемента массива, а потом снова взяли его адрес и присвоили его указателю *bar*. После этого получилось, что *bar* указывает уже не на первый, а на второй элемент массива *foo*, поэтому и конструкция “bar[1]“ указывает на третий элемент вместо второго.

Тех, кто ничего не понял из написанного на этой странице, можно успокоить: это довольно сложный материал, и он не входит в необходимый минимум знаний. Однако если вы с ним разберётесь, то получите мощный инструмент, который пригодится в том случае, если вам когда-нибудь доведётся столкнуться с низкоуровневым программированием.

Управляющие конструкции

В этом разделе мы рассмотрим специальные конструкции языка C, которые позволяют производить различные проверки, организовывать циклы и ветвления в логике программы.

Оператор “if”

Начнём с самого простого и наиболее часто используемого оператора. Оператор *if* позволяет провести проверку некоторых условий и выполнить определённые действия на основании результатов этой проверки. Сразу рассмотрим пример.

```
int foo = 5;
if (foo > 10)
{
	// do stuff
}
else
{
	// do other stuff
}
```

Как видите, после оператора *if* в круглых скобках следует проверяемое условие. Если результат проверки будет равен *true*, будет выполнен блок, идущий сразу после оператора *if* и выделенный фигурными скобками. Также после этого блока может быть размещено ключевое слово *else* и ещё один блок кода, который будет выполнен в том случае, если результат проверки окажется равным *false*. Блок *else* не является обязательным: если его нет и результат проверки равен *false*, блок, идущий после оператора *if,* просто не будет выполнен.

Если блок кода, который необходимо выполнить, состоит только из одной инструкции, фигурные скобки можно опустить и записать условие в виде:

```
if (something)
      foo();
else
      bar();
```

Если условие, которое нужно проверить, нельзя уместить в обычную бинарную логику *true/false*, можно объединить подряд несколько операторов *if*:

```
int D = b * b – 4 * a * c;
int x1, x2;

if (D < 0)
      // Complex numbers
else if (D == 0)
      x1 = x2 = -b / (2 * a);
else
{
      x1 = (-b + sqrt(D)) / (2 * a);
      x2 = (-b - sqrt(D)) / (2 * a);
}
```

Оператор “switch”

Бывают случаи, когда нужно сделать много однотипных проверок. Для этого можно использовать несколько операторов *if*, однако, эффективнее будет применить *switch*. Рассмотрим пример.

```
int foo = 5;

switch (foo)
{
      case 0:
            ZeroFunc();
            break;
      case 1:
            OneFunc();
            break;
      case 2:
            TwoFunc();
            break;
      default:
            DefaultFunc();
}
```

Здесь после оператора *switch* в скобках находится проверяемое выражение, а внутри блока *switch* расположены возможные результаты этой проверки. Каждый из

результатов обозначается ключевым словом *case* и двоеточием. После двоеточия может следовать сколько угодно инструкций, которые обязательно должны заканчиваться инструкцией *break*. Если не поставить *break*, будут выполнены все остальные инструкции из последующих блоков *case*. Инструкции из блока *default* выполняются в том случае, если ни один из блоков *case* не удовлетворил результату проверки.

Оператор “for”

Оператор *for* позволяет организовывать циклы. *Цикл* — это последовательность действий, выполняемых много раз с незначительными изменениями или над разными объектами. Как водится, начнём с примера.

```
int foo[6] = {4, 8, 15, 16, 23, 42};
int bar = 0;
int i;

for (i = 0; i < 5; ++i)
{
	bar += foo[i];
}
```

Сначала мы объявляем и инициализируем массив из шести элементов, затем объявляем переменную для хранения результата и *переменную-счётчик*. Счётчик используется для того, чтобы хранить количество выполненных итераций цикла.

Далее идёт сам цикл. Оператор *for* имеет следующую форму:

```
for (инициализация_счётчика; условие_выполнения; выражение)
{
}
```

Как видите, блок, заключённый в скобки, имеет три части, разделённые точками с запятыми. Каждая из этих частей может представлять собой абсолютно любое выражение. Первая часть выполняется один раз перед входом в цикл. Обычно её используют для присвоения начального значения переменной-счётчику (в нашем примере *i = 0;*). Вторая часть представляет собой условие выполнения цикла. Это условие работает как оператор *if*, который выполняется перед каждой новой итерацией. Если результат проверки равен *true*, итерация выполняется; если *false* — цикл завершается. В нашем примере мы проверяем значение счётчика *i < 5*. Третий блок выполняется *после* каждой итерации. Обычно этот блок используется для увеличения значения счётчика (в нашем примере *i++*).

Код, размещённый в фигурных скобках за оператором *for*, называется *телом цикла* и выполняется до тех пор, пока не будет произведён выход из цикла. Кстати, выход из цикла можно сделать и вручную, не дожидаясь выполнения условия, указанного в операторе *for*. Для этого в теле цикла нужно поместить оператор *break*. Например:

```
for (i = 0; i < 10; i++)
{
	if (red_alert)
		break;
	// Do normal stuff in a loop
}
```

Циклы часто используются для обработки массивов, при этом счётчик цикла используется как индекс элемента массива. Например, цикл в первом куске кода вычисляет сумму элементов массива *foo*, которая будет находиться в *bar* после завершения цикла.

Оператор “while”

Оператор *while* предоставляет нам ещё один способ организовать циклы. Он устроен проще, чем оператор *for* и может применяться в тех случаях, когда цикл не нуждается в счётчиках.

```
while (!SkyIsBlue())
{
	RemoveClouds();
}
```

Как можно понять из этого примера, цикл *while* выполняется до тех пор, пока выражение в скобках равно *true*.

Цикл *while* очень важен при программировании микроконтроллеров, так как именно с его помощью организуется главный цикл (подробнее о главном цикле — в первой лабораторной):

```
while (1)
{
	CheckButtons();
	GeneratePWM();
	GetADC();
	wdt_reset();
}
```

Функции

Функции — это очень важная часть любой программы на Си. С их помощью можно структурировать программу, выделив определённые блоки кода и обращаясь к ним по мере необходимости.

Функцией называется блок кода, имеющий имя и, возможно, возвращающий какое-то значение.

```
int add(int a, int b);

int main()
{
	int foo = add(5, 2);
	int bar = add(6, 8);

return 0;
}

int add(int a, int b)
{
	return a + b;
}
```

Разберёмся, что же здесь произошло. На первой строке, в области глобальных переменных, мы объявили *сигнатуру* функции. Сигнатура состоит из типа возвращаемого функцией значения, имени функции и списка параметров. Сигнатура функции объявляется для того, чтобы все функции в пределах одного файла знали о существовании друг друга. Для функции *main* сигнатуру можно не объявлять, так как она вызывается автоматически при запуске программы и вызывать её вручную крайне не рекомендуется (если очень хочется, то можно, однако, это сильно смахивает на индусский код).

После функции *main* следует *реализация* функции *add*, сигнатуру которой мы объявили ранее. Как видите, мы передаём в функцию две переменные типа *int*. Они называются *параметрами* или *аргументами* функции. Существует несколько способов передачи параметров. В нашем примере используется передача по значению. Это означает, что при входе в функцию создаются новые переменные с указанными именами, которым присваиваются переданные значения. По сути, параметры функции эквивалентны локальным переменным, объявленным в теле этой функции. Мы можем использовать их как обычные переменные, и даже изменять их значения. Хоть это и никак не отразится на значениях, переданных вызывающей функцией, изменять аргументы не рекомендуется хотя бы только потому, что это ухудшает читаемость кода и затрудняет его понимание.

Основная задача многих функций — обработать некую информацию, принятую в виде аргументов и вернуть результат. Функция может вернуть одно значение любого типа. Этот тип указывается в сигнатуре функции перед её именем. Возвращаемое значение — это ещё одна переменная, которую можно использовать наравне с остальными. Как видите, в функции *main* мы присваиваем результат выполнения функции *add* обычной переменной. Для возврата значения из функции служит ключевое слово *return*. Как только процесс выполнения программы подходит к этому оператору, производится выход из текущей функции и возврат в вызывающую функцию. В нашем примере происходит такая последовательность действий:

```
Объявление переменной foo →
вызов функции add с параметрами 5 и 2 →
вход в функцию add →
сложение аргументов →
return →
возврат в функцию main →
присваивание результата выполнения add переменной foo →
продолжение выполнения функции main.
```

Ключевое слово *return* можно использовать сколько угодно раз. Важно лишь помнить, что после возврата выполнение функции немедленно прекращается. Например, в следующем коде третий оператор *return* никогда не будет выполняться:

```
int foo(int bar)
{
	if (bar < 0)
		return 1;
	else
		return 2;

	return 3; // Will never get here.
}
```

Функция может и не возвращать никакого значения. В этом случае она объявляется с типом возврата *void* и не требует обязательного оператора *return* в конце.

```
void foo(int bar)
{
	if (bar > 0) {
		DoGreaterStuff();
		return;
	}

	DoLesserStuff();
} // We don’t need to return anything from void function
```

Стоит ещё раз повторить, что возврат в вызывающую функцию после завершения вызываемой происходит автоматически. Это значит, что если мы вызываем функцию *add* из функции *main*, нам не нужно ещё раз вызывать *main* в самом конце *add*. После завершения функции *add* выполнение программы продолжится с того самого места *main*, на котором оно прервалось.

Теперь рассмотрим ещё один способ передачи параметров — по указателю. В этом случае в функцию передаётся не значение переменной, а указатель на неё. Таким образом, внутри вызываемой функции мы можем изменить значение переменной, указатель на которую нам передали.

```
void foo(int* arg);

int main()
{
	int bar = 5;
	foo(&bar);
	// bar == 6 now.
	return 0;
}

void foo(int* arg)
{
	(*arg)++;
}
```

Мы объявили функцию как и раньше, однако, теперь передаём туда не значение переменной типа *int*, а указатель на эту переменную. Это даёт нам возможность разыменовать указатель в теле функции и получить доступ к исходному значению переменной. Есть несколько сценариев использования передачи параметров по указателю. Один из них — передача массивов. Массивы в С не могут быть переданы по значению. Чтобы передать массив в функцию, нужно использовать указатель.

```
void foo(int* arg);

int main() {
	int bar[3] = {1, 2, 3};
	foo(bar);
}

void foo(int* arg) {
	arg[0] = arg[1] + arg[2];
}
```

Мы воспользовались тем, что переменная массива является указателем на первый его элемент и передали этот массив по указателю в функцию *foo*.

Передачу аргументов по указателю можно также использовать в тех случаях, когда функция должна вернуть больше одного значения. Одно из значений можно возвращать с помощью оператора *return*, а для всех остальных объявить переменные в вызывающей функции и передать их по указателю в вызываемую функцию.

Символы и строки

Важной частью любой программы является вывод некоей информации. Это могут быть результаты расчётов, отчёт о текущем состоянии или просто отладочная информация. Обычно информация выводится в виде простого текста, но как этот текст представлен в исходной программе? В этом разделе мы попробуем разобраться в этом вопросе.

Символы

В стародавние времена, когда компьютеры были большими, а программистов было мало, основным «человеческим» языком, на котором говорили компьютеры, был английский. Чтобы выводить на экран или принтер какую-либо информацию, создатели компьютеров решили договориться о способе сопоставления числовых значений и печатных символов, которые бы им соответствовали (это и логично, так как компьютер знает только про числа, а не про какие-то там символы или строки). Так родился стандарт ASCII, который устанавливал соответствие между числами от *32* до *127* буквам латинского алфавита и некоторым служебным символам. Числа от 0 до 31 считались служебными и на экран не выводились, а числа от 128 до 255 резервировались для специальных символов. Таким образом, для хранения в памяти одного символа было достаточно одного байта, который, как известно, может принимать значения от 0 до 255. Именно поэтому однобайтовый тип переменных в C и называется *char* — от английского “character”. Всё это означает, что символы хранятся в памяти как обычные числа, и их интерпретация зависит только от того, как их применять. Вы вполне можете проводить небольшие вычисления с помощью типа *char*, и, в то же время, использовать этот тип как репрезентацию символа.

```
char foo = 32;
char bar = foo + 2;
```

В переменной *bar*, как вы уже поняли, будет находиться число 34. Вы можете использовать его как обычное число в своём коде. Однако если вы передадите это число в какую-нибудь функцию, которая выводит символы на экран, она интерпретирует число 34 как символ “C”. Вам необязательно знать числа, которые соответствуют символам. Можно написать, например, такой код:

```
char foo = 'A';
char bar = foo + 2;
```

Компилятор автоматически интерпретирует символ 'A' в значение *32*, поэтому этот отрывок кода будет равнозначен предыдущему.

После принятия стандарта ANSI все жили долго и счастливо, пока в один прекрасный день внезапно не выяснили, что далеко не все люди на планете говорят по-английски. Вторым неожиданным открытием был тот факт, что верхних 127 символов,

зарезервированных стандартом как специальные, не хватит для того, чтобы представить все символы из других языков. Немного поразмыслив, программисты решили пойти на компромисс и создали концепцию кодовых страниц. Фокус заключался в том, что каждый язык должен сам определять, какие числа из диапазона 127-255 соответствуют каким символам из этого языка. Это значит, что значение 150 могло означать один символ в русской кодировке и совсем другой символ в японской. В памяти символы всё так же продолжали храниться в виде чисел от 0 до 255, однако изменился способ, с помощью которого интерпретировались эти числа. Теперь для отображения символов какого-то специфического языка нужно было указывать соответствующую кодовую страницу, иначе на экране появлялись невменяемые «кракозябры». При этом числа от 32 до 127 всё также были зарезервированы для латиницы, независимо от кодовой страницы.

К счастью, в наши дни таких проблем практически не осталось благодаря новому стандарту Unicode, уверенно шагающему по планете. Согласно этому стандарту, для хранения символа отводится не один, а целых два байта (значения от 0 до 65535), позволяя дать каждому символу из любого мыслимого языка свой уникальный номер. С повсеместным введением Unicode исчезнет само понятие кодовых страниц и, будем надеяться, проблем, с ними связанных (по крайней мере, до тех пор, пока человек не познакомится с остальными обитателями Галактики и обилием их языков).

Однако не всё так радужно с юникодом, как нам хотелось бы. Конечно, в новых языках программирования, таких как C# или Java, символы по умолчанию являются юникодными, и большинство адептов этих языков искренне не понимают сути проблем с кодировками. К сожалению, в случае программирования микроконтроллеров на C всё обстоит не так гладко. Во-первых, язык C разрабатывался во времена, когда юникода ещё не было даже в планах. Конечно, современные компиляторы поддерживают использование юникода в программах на C, однако, это скорее костыли, нежели нормальный инструмент. Во-вторых, сами контроллеры накладывают на нас ограничения объёмами своей памяти. Часто бывает так, что памяти не хватает даже для скомпилированного кода, который приходится оптимизировать так, чтобы он влез в отведённые килобайты. На этом фоне выделение двух байт на каждый символ выглядит просто непростительно. По этим двум причинам нам придётся использовать старые добрые ANSI кодировки (особенно попотеть придётся при выводе русских символов на LCD-дисплей).

Строки

Вполне логично, что возможности по работе с текстом в C не ограничиваются отдельными символами. Было бы довольно изнурительно выводить на экран информацию по одной букве. Именно для этого и была введена концепция *строки*. Строка — это просто последовательность символов (значений типа *char*). В конце этой последовательности обязательно должен стоять *нулевой символ* — '\0'. Начало строки представляется указателем на её первый символ. Затем последующие символы считываются автоматически до тех пор, пока не встретится нулевой символ. Следующий код иллюстрирует работу со строками.

```
char s1[13] = "Hello world!";
char s2[13] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
char s3[] = "That's weird!";

void TransmutateString(char* str) {
      int i;

      for (i = 0; i < strlen(str); i++)
            str[i] += 2;
}

void main() {
      TransmutateString(s1);
}
```

В этом коде показано несколько приёмов. Во-первых, это различные способы задания строк. Переменной s1 строка присваивается с помощью двойных кавычек. В переменной s2 мы явным образом задаём массив символов, составляющих ту же строку. Обратите внимание, что в этом случае мы вручную добавляем нулевой символ, тогда как при использовании двойных кавычек компилятор делает это автоматически. Также, и в первом и во втором случае мы явным образом указываем размер массива, причём этот размер на единицу больше, чем длина строки (именно из-за необходимости хранить нулевой символ). В третьем же случае мы не указываем размер массива, компилятор делает это за нас при сборке программы.

Мы также написали функцию *TransmutateString*, которая производит некоторые операции над каждым символом строки. Для передачи строки в виде параметра мы должны объявить этот параметр как указатель типа *char*. Если вы вспомните связь между массивами и указателями, то поймёте, что строка ничем не отличается от массива, поэтому мы можем обратиться к ней просто по имени переменной, например, *s1*.

Ну и наконец, функция *strlen*, которая используется в *TransmutateString,* является библиотечной и возвращает длину указанной строки без учёта нулевого символа.

Препроцессор

Помимо стандартных средств программирования, присущих большинству языков, в С имеется ещё один мощный инструмент — препроцессор. Он является в некотором роде метаязыком: обычные конструкции С контролируют ход выполнения программы, а препроцессор определяет то, как будет выглядеть исходный код программы ещё до компиляции. К примеру, на этапе компиляции при выполнении заданных заранее условий могут динамически включаться или отключаться целые блоки кода.

Препроцессор можно задействовать, используя *директивы*. *Директивы препроцессора* похожи на обычные ключевые слова, отличить их можно по символу октоторпа (“#”). Далее мы рассмотрим наиболее часто применяющиеся директивы.

#define

Директива *#define* позволяет создавать *макросы*. Макрос — это некий идентификатор, в соответствие которому поставлено какое-то значение. При компиляции программы все макросы в файлах заменяются на их непосредственные значения, и лишь затем начинается сам процесс компиляции. Рассмотрим пример.

```
#define THREE      3

int main()
{
      int foo = THREE; // Translates into “int foo = 3;”

      return 0;
}
```

Здесь мы создали самый простой макрос с именем *THREE*, который имеет значение *3*. Теперь препроцессор, который запускается перед компилятором, будет заменять все вхождения макроса *THREE* на цифру *3*. Здесь стоит заметить несколько важных моментов.

1. Для имени макроса действуют те же правила, что и для имён переменных. Однако для удобства макросам обычно дают имена в верхнем регистре, чтобы не путать их с обычными функциями и переменными.
2. Макрос состоит из трёх частей: директивы *#define*, имени макроса и подставляемого значения. Каждая из частей отделяется от других одним или несколькими пробелами или символами табуляции. Роль играют только первые два пробела, все остальные пробелы учитываются как значение для подстановки. Например, вместо следующего макроса будет подставляться значение “one two three”:

```
#define ONE_TWO_THREE one two three
```

3. После определения макроса точка с запятой **не ставится**. Символом окончания подстановочного значения является перенос строки.

С помощью макросов можно объявлять числовые константы.

```
#define ARRAY_SIZE        10

int main()
{
      int foo[ARRAY_SIZE];
      int i;

      for (i = 0; i < ARRAY_SIZE; ++i)
            foo[i] = i;

      return 0;
}
```

В этом примере мы используем размер массива в двух случаях: при объявлении самого массива и в цикле *for*. Если бы мы указали размер явным образом и в какой-то момент захотели изменить количество элементов массива, нам пришлось бы изменять значение сразу в двух местах. Очень часто бывают случаи, когда одно и то же значение используется в разных местах кода, и использование макросов позволяет быстро изменить это значение и не допустить ошибок.

Макросы также имеют и более сложные применения.

```
#define SB(a,b)    (a) |= 1 << (b)

int main()
{
      int foo = 2;

      SB(foo, 2); // Same as writing “foo |= 1 << 2;”

      return 0;
}
```

В этом примере мы объявили макрос, который может принимать аргументы. В этом плане макросы могут быть похожи на функции, за тем лишь исключением, что аргументы макросов не имеют типа и, по сути, являются обычным текстом для подстановки. Как видно из примера, макрос устанавливает в единицу указанный бит указанной переменной.

#ifdef, #else и #endif

Директива *#ifdef* напоминает обычное условие *if*. Эта директива позволяет проверить, определён ли указанный макрос и, если это так, включить определённый блок кода.

```
#define FOO  1

int main()
{
#ifdef FOO
      int foo = 5;
#endif

#ifdef BAR
      int bar = 5;
      DestroyTheWorld();
#else
      int bar = 10;
#endif

      int res = foo + bar;      // ret == 15

      return 0;
}
```

Здесь мы сначала определяем макрос *FOO*, а затем в функции *main* проверяем его наличие с помощью директивы *#ifdef*. Так как макрос определён, блок кода, заключённый между директивами *#ifdef* и *#endif* будет обработан компилятором.

Далее мы проверяем, объявлен ли макрос *BAR*. Так как никто его не объявлял, компилятором будет обработан блок кода, заключённый между директивами *#else* и *#endif*, в то время как блок между *#ifdef* и *#else* будет проигнорирован.

#undef

Существует также директива *#undef*, которая позволяет отменить объявление макроса.

```
#define FOO  1

int main()
{
#ifdef FOO
       int foo = 5;
#endif

       foo++;

#undef FOO

       foo--;

#ifdef FOO
       DestroyTheWorld();
#endif

       return 0;
}
```

Во время первой проверки макрос *FOO* ещё определён, поэтому объявление переменной *foo* будет обработано компилятором. Затем следует директива *#undef* и во время второй проверки макрос *FOO* будет уже не определён и соответствующий блок кода будет проигнорирован.

Структура программы на С

Во всех примерах, которые мы до сих пор рассматривали, предполагалось, что вся программа состоит только из одного файла, в котором находится весь код. Этот сценарий имеет право на жизнь, однако, почти никогда не применяется. Программа может быть очень сложной и состоять из сотен функций. Понятно, что в этом случае хранить весь код в одном файле — чистое безумие. В этом разделе мы рассмотрим способы, с помощью которых код программы можно разместить в разных файлах.

Два типа файлов: *.h и *.c

В обычную программу на С, как правило, входят два типа файлов: *заголовочные файлы* (с расширением *.h) и *файлы кода* (с расширением *.c). Обычно, эти файлы используются в паре. Например, если у вас есть файл с кодом под названием *“functions.c“* и вы хотите использовать функции из этого файла в других местах, вам также нужно создать заголовочный файл *“functions.h”* (на самом деле, имена могут быть любыми, но логичнее давать файлам одинаковые имена).

Заголовочные файлы являются своего рода картой или «оглавлением», в котором перечислены «возможности», реализуемые соответствующим файлом кода. В разделе, посвящённом функциям, мы узнали, что для каждой функции должен быть объявлен прототип, определяющий её имя, параметры и тип возвращаемого значения. Если вы хотите использовать функцию из файла кода в других местах, то прототип этой функции помещается не сам файл кода (как в предыдущих примерах), а в соответствующий заголовочный файл.

Директива #include

Итак, мы разделили прототипы функций и их реализацию, но что дальше? Теперь мы должны включить заголовочный файл во все файлы, в которых мы собираемся использовать его функции. Именно для этого и служит директива *#include*. При обработке препроцессором вместо этой директивы подставляется содержимое файла, указанного в этой директиве. Рассмотрим пример.

Functions.h:

```
#ifndef _FUNCTIONS_H
#define _FUNCTIONS_H

int foo(int arg1, int arg2);
int bar();
void baz();

#endif
```

Functions.c:

```
#include “functions.h”

int foo(int arg1, int arg2) {
      return arg1 + arg2;
}
int bar() {
      return 42;
}
void baz() {
      sleep(1000);
}
```

main.c

```
#include “functions.h”

int main()
{
	int res = foo(2, 5);
	int answerToAllQuestions = bar();

baz();

return 0;
}
```

Итак, в этом примере есть три файла: *functions.h*, *functions.c* и *main.c*. Подразумевается, что в *functions.c* находятся некоторые служебные функции, которые мы хотим использовать в основном файле *main.c*.

В заголовочном файле *functions.h* расположены прототипы служебных функций. Благодаря этому файлу компилятор будет «знать», какие функции ему доступны. Обратите внимание на хитрую конструкцию из директив препроцессора, в которую обёрнуто содержимое заголовочного файла. Эта конструкция нужна для того, чтобы при включении заголовочных файлов в разные места компилятор обработал его содержимое только один раз. Этот нюанс довольно сложно объяснить простым языком, поэтому, нужно просто запомнить, что содержимое заголовочного файла всегда надо оборачивать в подобную конструкцию (имя макроса может быть любым, но обязательно уникальным и неповторяющимся в других заголовочных файлах).

В файле кода *functions.c* мы первым делом включаем наш заголовочный файл. Таким образом, мы, по сути, вставляем в самое начало файла наши прототипы, определённые в заголовочном файле. Далее мы просто реализуем объявленные функции, написав для них необходимый код.

Файл *main.c* является основным для нашей программы, так как в нём расположена функция *main*. Мы хотим использовать в ней наши служебные функции, поэтому включаем заголовочный файл и сюда. Это, опять же, равносильно копированию прототипов функций и вставке их в начало файла. Теперь, имея объявленные прототипы, мы можем использовать функции из *functions.c* так, как если бы они находились в самом файле *main.c*. Всё, что нужно — это поместить в *main.c* прототипы нужных функций, а о поиске непосредственного кода этих функций позаботится компилятор.

Создание сложной структуры

В заголовочных файлах можно размещать не только прототипы функций, но и объявления макросов, структур и других типов данных. Кроме этого, заголовочные файлы можно включать не только в файлы кода, но и в другие заголовочные файлы.

Macro.h

```
#ifndef _MACRO_H
#define _MACRO_H

#define UINT unsigned long int
#define UCHAR	unsigned char

#endif
```

Functions.h

```
#ifndef _FUNCTIONS_H
#define _FUNCTIONS_H

#include “Macro.h”

UINT UnsignedAdd(UINT a, UINT b);
UCHAR UnsignedAddChar(UCHAR a, UCHAR b);

#endif
```

Functions.c

```
#include “functions.h”

UINT UnsignedAdd(UINT a, UINT b)
{
	return a + b;
}

UCHAR UnsignedAddChar(UCHAR a, UCHAR b)
{
	return a + b;
}
```

main.c

```
#include “Macro.h”
#include “Functions.h”

int main()
{
	UINT resI = UnsignedAdd(17000 + 18000);
	UCHAR resC = UnsignedAddChar(127 + 129);

	return 0;
}
```

В этом примере мы создали макросы *UINT* и *UCHAR* для того, чтобы сократить длинную запись беззнаковых типов *int* и *char*. Затем мы включили заголовок *Macro.h* в другой заголовок *Functions.h*, в котором объявили функции, использующие новые «типы». Далее мы пошли по уже проторённой дорожке: реализовали служебные функции в *functions.c* и использовали их в *main.c*. Строго говоря, мы могли бы не включать *Macro.h* в *main.c*, так как этот заголовок уже содержится во включённом *functions.h*. Однако мы сделали это для наглядности, так как не всегда бывает удобно держать в голове структуру заголовков и вещи, которые в них объявлены. Компилятор же проигнорирует «лишнее» включение именно из-за той самой препроцессорной «обёртки».

Заключение

На этом мы закончим краткий обзор языка C. Безусловно, за бортом осталось много неосвещённых тем. Некоторые из них будут обсуждаться в описаниях к лабораторным работам, а некоторые, наиболее сложные и не входящие в обязательную программу, можно изучить самостоятельно с помощью книг из списка литературы. Напоследок хотелось бы сделать несколько небольших, но важных замечаний.

Вопреки расхожему мнению, недостаточно написать код, который просто работает. Всегда нужно стремиться к коду, который можно легко понять и изменить. Один из гуру программирования сказал по этому поводу: «Пишите код так, как если бы после вас его поддерживал психопат, склонный к насилию и знающий, где вы живёте». Хороший код

складывается из многих элементов. Вот несколько советов, которые могут вам пригодиться:

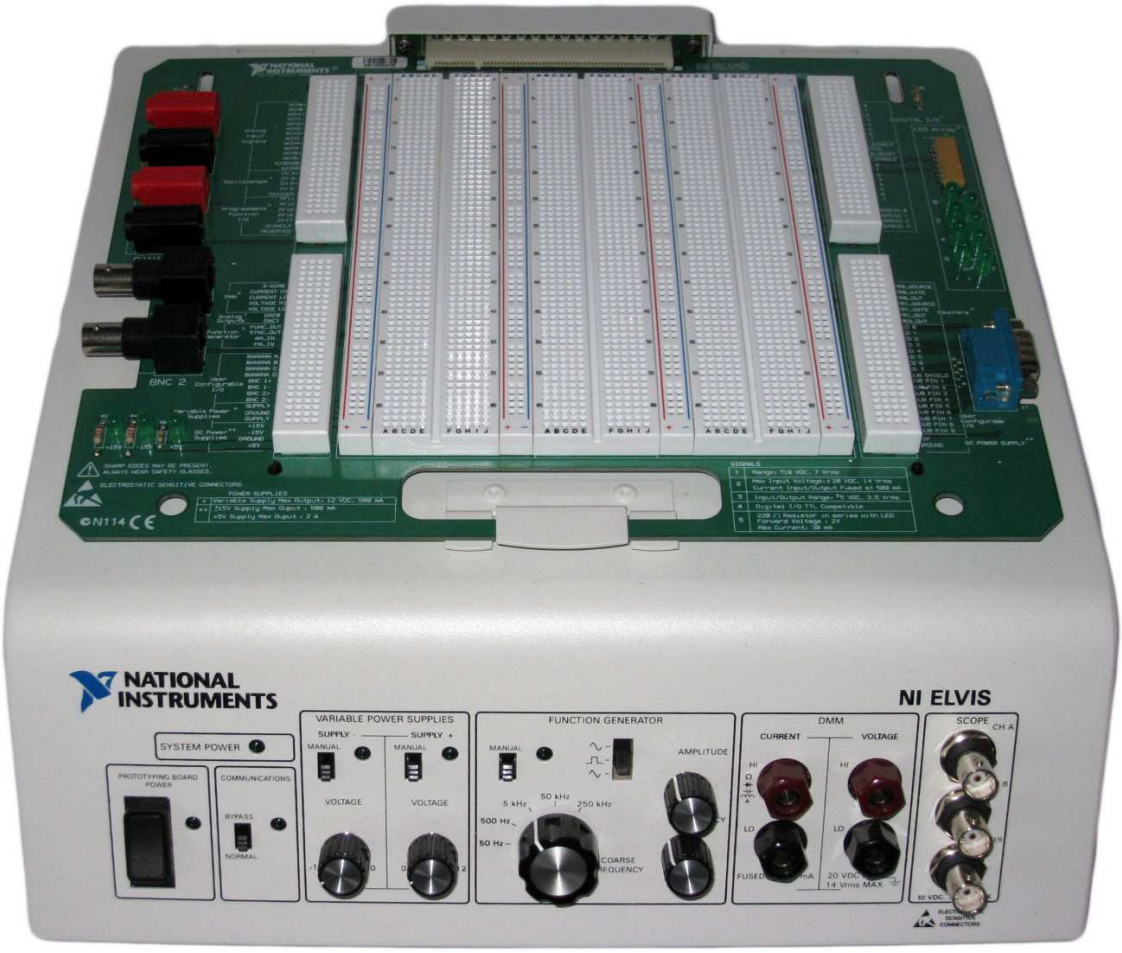
1. Пишите комментарии. Это не значит, что нужно комментировать каждую строку и подробно описывать тривиальные вещи. Однако комментарии должны давать общее представление о том, что делает код, и как он работает. Не стесняйтесь писать длинные комментарии. Если они действительно содержат полезную информацию, вам только скажут «спасибо».
2. Разбивайте код на функции. Если вы видите, что какой-то участок кода решает вполне определённую и логически независимую задачу, этот участок вполне можно выделить в функцию. Такой подход упрощает понимание программы и позволяет быстрее вносить в неё изменения. Если вы опасаетесь за издержки производительности, связанные с вызовом функций, вы можете объявлять их с ключевым словом *inline*. В этом случае компилятор будет подставлять тело функции в места, где она вызывается (аналогично макросам). Однако в большинстве случаев такой подход излишен, так как компилятор самостоятельно проводит необходимые оптимизации.
3. Используйте локальные переменные, когда это возможно. Глобальные переменные ухудшают восприятие кода и повышают вероятность появления трудноуловимых багов. Кроме этого, глобальные переменные постоянно занимают отведённую им память, что в условиях ограниченности ресурсов микроконтроллера является серьёзным недостатком. В большинстве случаев глобальные переменные легко заменяются передачей переменных в качестве аргументов функций.
4. Применяйте константы для часто используемых значений. Мы уже обсуждали этот вопрос в разделе, посвящённом макросам. К этому можно также добавить тот факт, что именованные константы дают дополнительные подсказки о том, что делает конкретный код. Однако не стоит увлекаться этим правилом: однажды в одном из проектов я встретил константу “THOUSAND = 1000”. Для меня до сих пор остаётся загадкой, что имел в виду автор. Возможно, он хотел предусмотреть в своей программе абсолютно всё, вплоть до изменения фундаментальных законов мироздания.
5. Форматируйте код. Применяйте отступы для блоков кода, заключённых в фигурные скобки. Код с отступами читается гораздо легче, чем «спагетти» из операторов, функций и управляющих конструкций (а если там, к тому же, нет ни одного комментария, то вообще замечательно). Всегда используйте один и тот же стиль форматирования. Например, существуют разные стили оформления блоков с кодом: кто-то ставит открывающую фигурную скобку на одной строке с ключевым словом или функцией, а кто-то — на следующей. Это никоим образом не влияет на результирующий код, и каждый волен использовать стиль, который ему больше по душе. Самое главное — придерживаться этого стиля во всех случаях. В этом пособии мы смешиваем разные стили, однако, наша карма остаётся чистой, так как мы делаем это только для того, чтобы оптимально разместить блоки кода на печатных страницах. Не берите с нас пример, вам экономить нечего.

Описание лабораторного стенда

Стенд, на котором вам предстоит выполнять все лабораторные работы в этом курсе, состоит из двух частей: платформы NI Elvis и набора модулей для работы с контроллером ATMega16. В следующих разделах мы подробно рассмотрим каждый из этих компонентов.

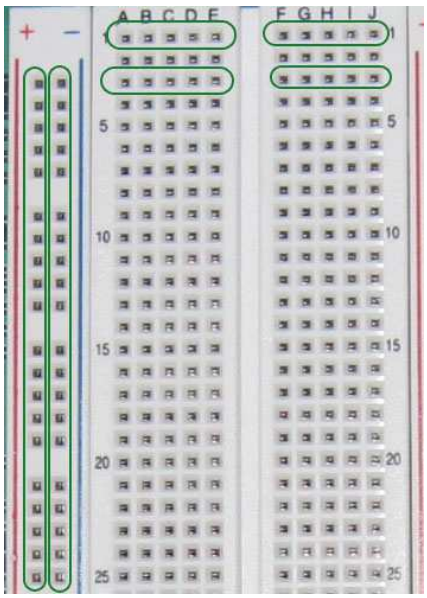
Платформа NI Elvis

Данная платформа представляет собой макетную плату, интегрированную с некоторыми интрументами.



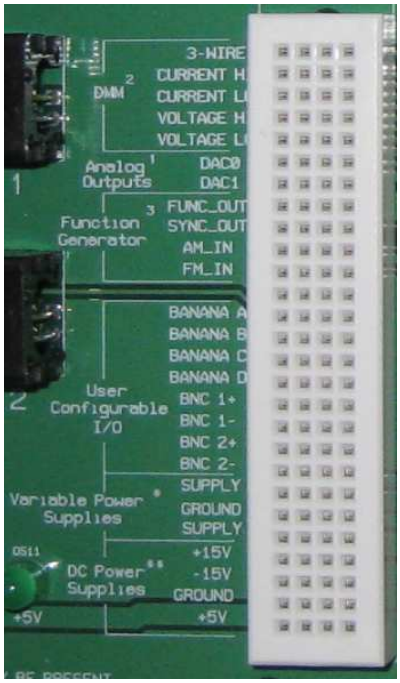
На фотографии представлен общий вид устройства. На передней панели расположены элементы управления и входы/выходы основных инструментов: источника питания, генератора импульсов, мультиметра и осциллографа. Эти выводы также продублированы на макетной плате, поэтому инструменты можно подключать прямо к схеме с помощью обычных перемычек. Платформа подключается к компьютеру, на котором можно получить доступ ко всем её возможностям с помощью фирменных программ NI.

Рассмотрим подробнее макетную плату. Контакты для подключения компонентов разбиты на группы, которые закорочены между собой.



Зелёным на фотографии отмечены группы контактов, представляющих одну логическую точку на электрической схеме. Шины питания (с красными и синими линиями) закорочены по всей высоте, поэтому достаточно подать питание лишь на один контакт. Каждая горизонтальная группа из пяти контактов на основных участках платы также представляет собой одну логическую точку.

По бокам от основной платы расположены четыре дополнительных участка, соединённых с дополнительными разъёмами или инструментами.



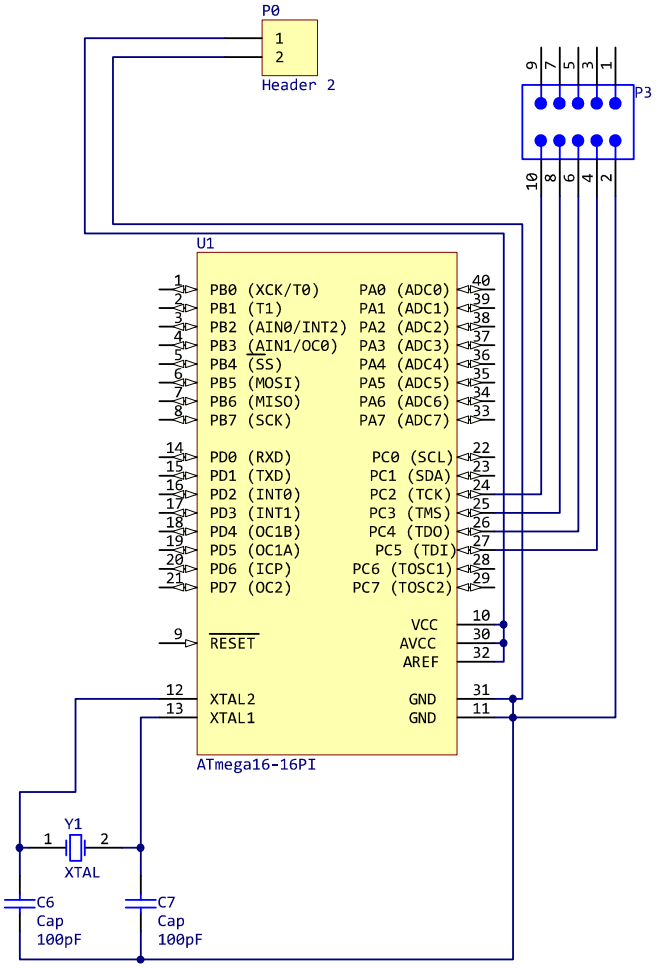
На фотографии показан нижний левый блок, контакты которого соединены с мультиметром, генератором импульсов и дополнительными разъёмами. Также в этом блоке находится важная группа “DC Power Supplies”, с контактов которой берётся напряжение питания для всей схемы. Именно эти контакты нужно соединять перемычками с вертикальными шинами для подачи на них питания и земли.

Набор модулей

Чтобы не тратить время на сборку сложных схем на каждой лабораторной работе, мы вынесли основные компоненты стенда в отдельные модули. Каждый из модулей отвечает за определённую функцию: главный модуль, модули кнопок и светодиодов и т. д. Модуль включает всю обвязку, необходимую для работы каждого из компонентов. Всё, что нужно сделать при проведении лабораторной работы — это правильно соединить модули на макетной плате Elvis. Однако это не значит, что модули должны быть для вас «чёрными ящиками». Единственная цель их существования — сэкономить время на сборке схемы, а значит, для успешного завершения курса вам придётся досконально изучить принцип работы всех компонентов, включённых в модули (на самом деле, там нет ничего сложного, по крайней мере, для тех, кто сдал электронику хотя бы на три).

Главный модуль

Начнём с описания главного модуля, на котором расположен сам микроконтроллер ATMega16 и несколько компонентов, необходимых для его работы.

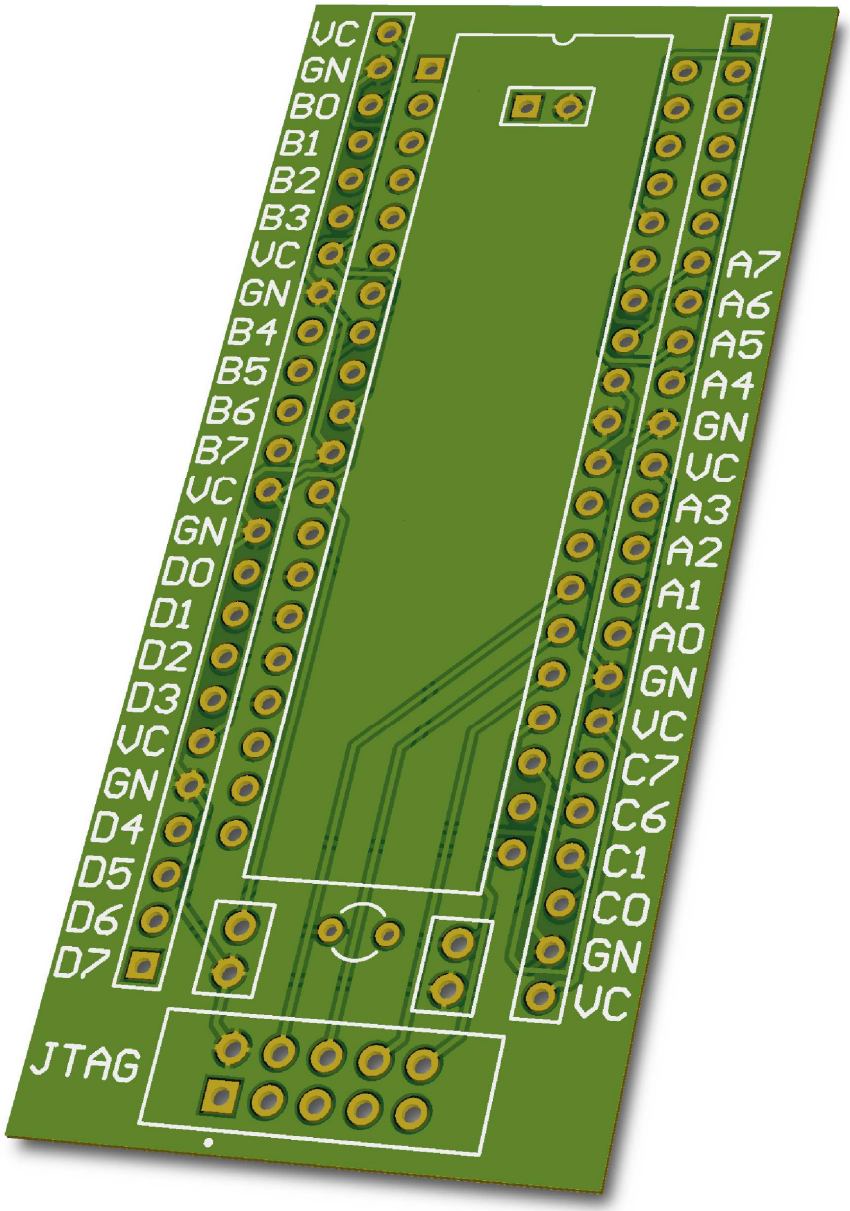


На рисунке приведена принципиальная схема главного модуля. Сверху на схеме расположен разъём питания, который соединён с выводами линий +5V и земли микроконтроллера. Также наверху расположен разъём для подключения программатора. Он подсоединён к соответствующим выводам порта C.

Внизу схемы можно увидеть кварцевый резонатор, подключённый к выводам XTAL1 и XTAL2 микроконтроллера. Кварцевый резонатор используется для задания системного тактового сигнала, который определяет быстродействие контроллера. В данном модуле используется кварц на 4 МГц.

Стоит также отметить, что кварцевый резонатор — это не единственный способ тактирования контроллера. ATMega16 имеет встроенный RC-осциллятор, позволяющий получить частоту до 8 МГц без использования внешнего кварца. Однако встроенный осциллятор не отличается высокой точностью, поэтому для случаев, когда необходима точная частота, стоит применять внешний кварц. Кроме этого, есть возможность получить тактовый сигнал не от кварца, а от тактового генератора, подключив его выход к XTAL1.

Все остальные выводы микроконтроллера выведены на коннекторы модуля. Посмотрим теперь на внешний вид платы модуля.



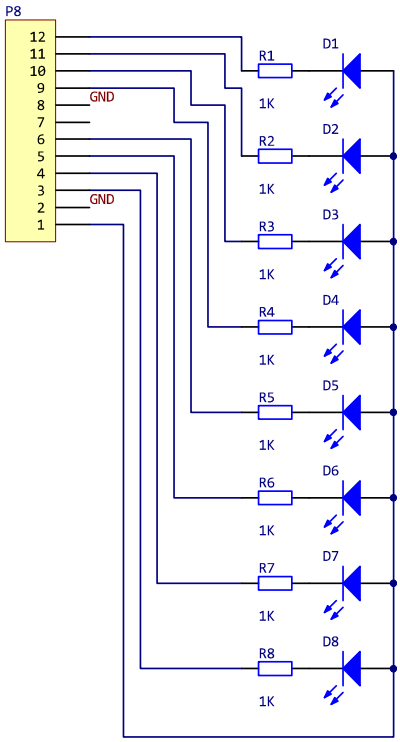
Как видно из картинки, сам микроконтроллер расположен в центре модуля, внизу находится разъём программатора, а по бокам размещены коннекторы, с помощью котрых модуль подключается к макетной плате. Для удобства подключения модулей выводы коннекторов расположены по следующей схеме: вывод питания (VC); вывод земли (GN);

4 вывода порта. Выводы портов идут по порядку за исключением порта C: в нём выведены только выводы PC0, PC1, PC6 и PC7. Остальные выводы используются для подключения программатора.

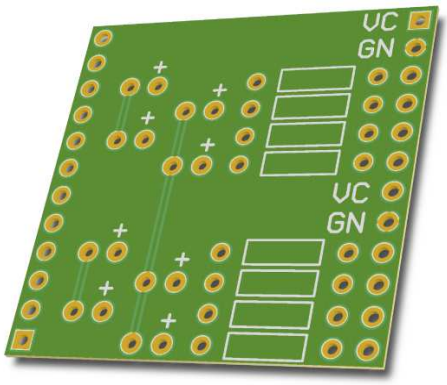
При подключении главного модуля к макетной плате нужно следить, чтобы два коннектора, расположенных по центру, подключились к шинам питания и земли Elvis.

Модуль светодиодов

В наборе модулей имеется плата с 8 светодиодами. Рассмотрим её схему.



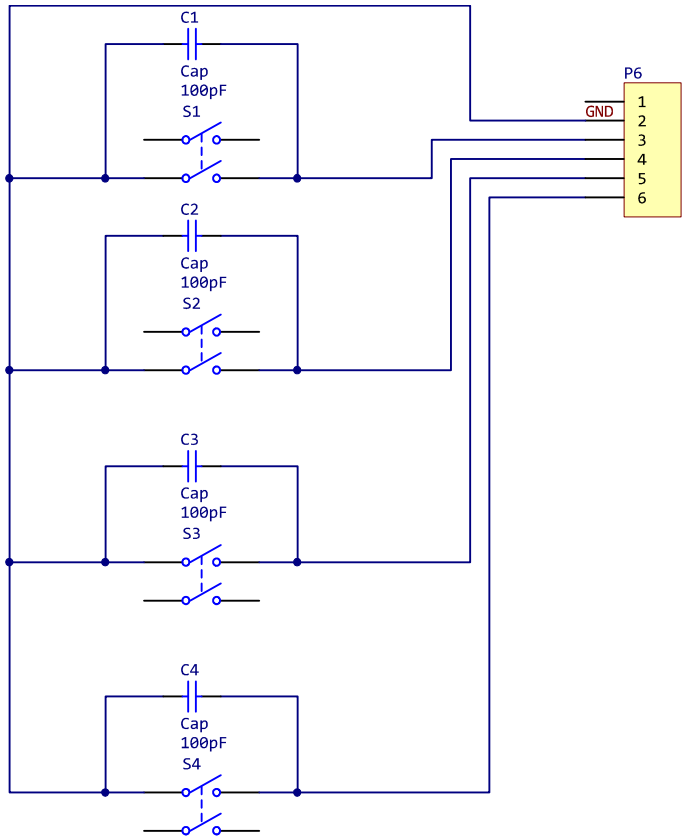
Как видно из схемы, модуль имеет 8 светодиодов, подключённых к коннектору через токоограничивающие резисторы. Подробнее схема подключения светодиодов будет рассмотрена в лабораторной работе №1.



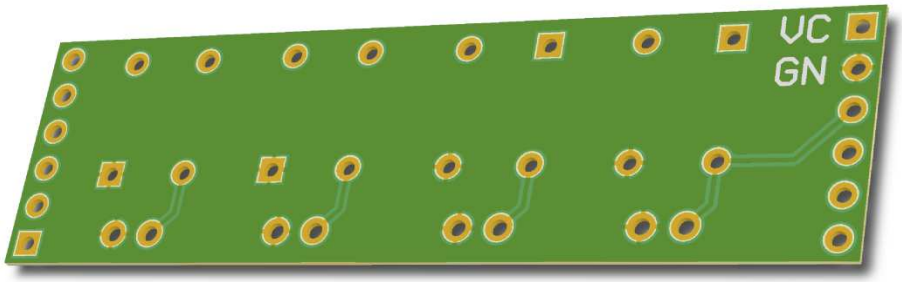
Модуль светодиодов можно подключить к любому порту, имеющему 8 пинов (можно, конечно, подключить к порту C и к половине порта A, но это сильно усложнит управление диодами).

Модуль кнопок

Для добавления интерактивности устройству, собранному на микроконтроллере, можно использовать кнопки. Ниже приведена схема модуля, содержащего 4 тактовых кнопки.



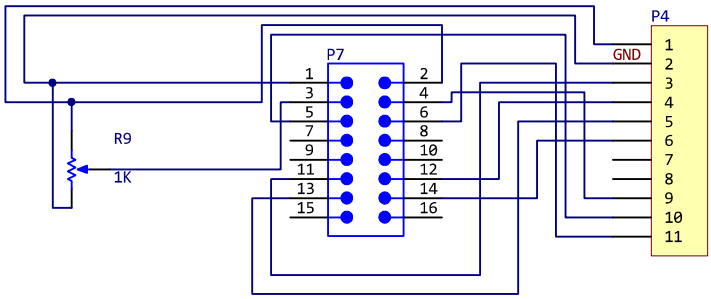
В схеме нет ничего необычного: тактовые кнопки подключены одним выводом к коннектору порта, а другим — к земле. К каждой кнопке дополнительно подключён конденсатор для предотвращения «звона». Подробнее об этой схеме подключения написано в описании к лабораторной работе №1.



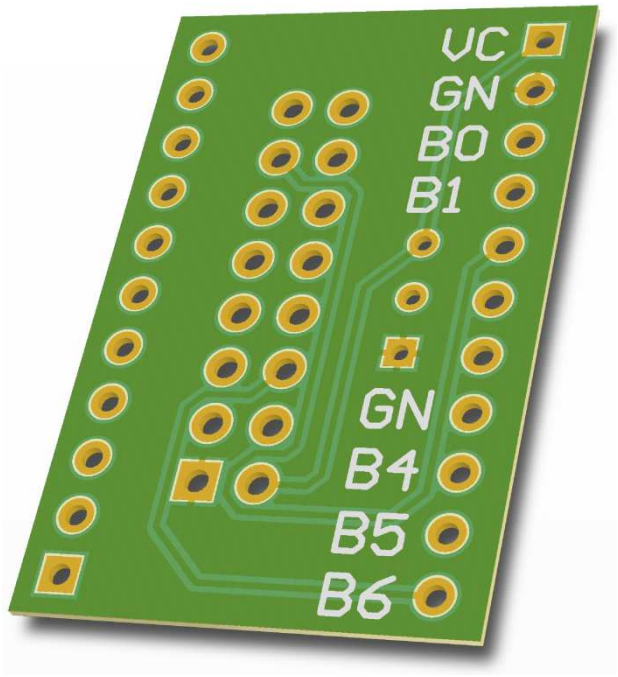
Плату с кнопками можно подключить к какому угодно порту, так как она занимает всего 4 вывода.

Модуль LCD-дисплея

Самый интересный компонент во всём курсе — это, конечно, LCD-дисплей. С его помощью можно организовать вывод любой информации, а самые продвинутые смогут даже сделать простенькую анимацию. У дисплея есть свой собственный контроллер, который управляет жидкокристаллической матрицей и предоставляет параллельный интерфейс. Для подключения ATMega16 к этому интерфейсу и используется данный модуль.



На плате расположен коннектор для подключения к Elvis, а также коннектор для подключения шлейфа LCD-дисплея. Кроме этого, на плате есть потенциометр, с помощью которого можно регулировать контрастность дисплея (если вместо потенциометра стоит перемычка, знайте, что человеку, который паял модуль, было лень, и вам теперь придётся смириться с максимальной контрастностью).



Для управления LCD-дисплеем из программы микроконтроллера применяется отдельная библиотека. В качестве параметров для этой библиотеки нужно указать порт, к которому подключен дисплей. По умолчанию указан порт B, и именно поэтому на плате явно написаны выводы, к которым её нужно подключать. Однако если вы чувствуете, что готовы к приключениям, эту настройку всегда можно изменить и подключить дисплей к какому угодно порту (если чувствуете себя супергероем, подключите дисплей к «половинному» порту C и половине порта A).

Лабораторная работа №1

В этой лабораторной работе мы более подробно рассмотрим некоторые элементы микроконтроллера и научимся выполнять простые действия с портами ввода-вывода.

Порты ввода-вывода

Как уже было сказано во вступлении, порты ввода-вывода представляют собой группу пинов[1] микроконтроллера, которыми можно управлять по собственному усмотрению. Все порты контроллера ATMega16 имеют по 8 пинов, по количеству бит в байте. Таким образом, состояние пинов каждого порта помещается в один однобайтовый регистр. Каждый пин может работать в двух режимах: ввод или вывод. Кроме этого, почти за каждым пином закреплена какая-нибудь специальная функция. Этими функциями можно управлять с помощью соответствующих регистров, которые будут рассмотрены позднее.

Микроконтроллер — это цифровое устройство, а значит, его пины могут находиться только в трёх состояниях: логическая «1», логический «0» и неопределённое состояние. Каждому из этих состояний соответствует диапазон напряжений, причём эти напряжения могут отличаться в зависимости от режима работы порта.

Режим работы	**Параметр**	**Условие**	**Мин. напряжение, В**	**Макс. напряжение, В**
Ввод	Низкий уровень	Vcc = 2.7 – 5.5 В	-0.5	0.2 * Vcc
	Высокий уровень		0.6 * Vcc	Vcc + 0.5
Вывод	Низкий уровень	Vcc = 5 В	—	0.7
	Высокий уровень		4.2	—

Если порт используется в режиме GPIO (General Purpose Input/Output, ввод-вывод общего назначения), для его управления применяются три регистра: *PORTx*, *DDRx* и *PINx*. Буква *x* в названиях регистров обозначает любой из четырёх портов контроллера, то есть, на её место можно подставить букву “A” и получить регистры для управления портом A: *PORTA*, *DDRA*, *PINA*.

Биты регистра *PORTx* отвечают за чтение и запись данных в случае, если соответствующий пин контроллера сконфигурирован для работы на вывод. Если же пин сконфигурирован для работы на ввод, данный регистр управляет включением или отключением подтягивающего резистора[2].

Регистр *DDRx* контролирует режим работы отдельных пинов порта. Если какой-то бит регистра *DDRx* имеет значение 0, соответствующий пин будет работать на ввод. Если же бит имеет значение 1, пин работает на вывод.

---

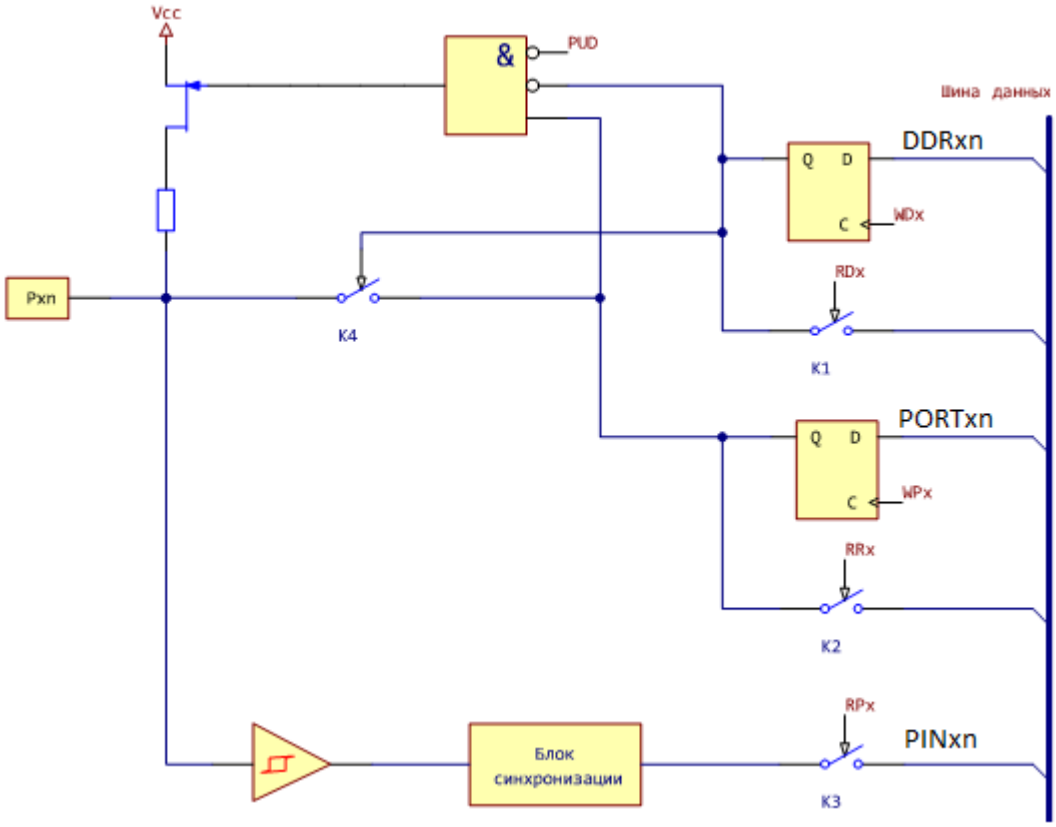
[1] Здесь и далее в отношении физических выводов микроконтроллера (ножек) будет применяться термин «пин». Это сделано во избежание параллелизма при одновременном использовании терминов «вывод микроконтроллера» и «вывод» в смысле режима работы порта.

[2] Подтягивающие резисторы мы рассмотрим через страницу. Ради интереса, попробуйте пока сами понять, для чего он нужен.

Регистр *PINx* позволяет получить доступ к текущему значению выбранного пина в независимости от того, в каком режиме данный пин работает. Обычно этот регистр используется только в режиме работы на ввод для получения текущего значения на пине.

Каждый пин любого порта может быть независимо сконфигурирован для работы на ввод, вывод, или для использования специальной функции. Например, можно использовать пины PB0[1] и PB1 для вывода, PB4 и PB5 для ввода, а PB2 и PB3 для работы с внешним прерыванием и таймером соответственно.

Чтобы лучше понять, как работают порты ввода-вывода, рассмотрим рисунок.



На этом рисунке представлена немного упрощённая принципиальная схема одного пина. Слева обозначен физический вывод контроллера Pxn, справа — шина данных (см. общую схему внутренностей контроллера во вступлении).

[1] Pxn — ещё одно название-шаблон, которое обозначает отдельный пин в каком-либо порту. x — это буква, обозначающая порт, а n — номер пина.

Рассмотрим верхний D-триггер. Он хранит значение бита n регистра DDRx. Допустим, мы хотим сконфигурировать пин 1 порта A для работы на вывод. Для этого нам надо записать единицу в соответствующий бит регистра DDRx. Мы можем использовать следующий код:

```
DDRA |= (1 << PA1);
```

При этом происходят следующие вещи:

- На вход “D” верхнего D-триггера поступает логическая «1»
- На вход “C” этого же триггера поступает импульс, приводящий к запоминанию триггером значения на входе “D”.
- С этого момента на выходе триггера “Q” будет находиться логическая «1» до того момента, как будет произведена следующая запись в этот бит.

Считать значение этого же бита можно с помощью кода:

```
if (DDRA & (1 << PA1))
      DoSomething();
```

При этом ключ K1 замкнётся и значение бита попадёт в шину данных.

По такому же принципу работает и второй D-триггер, отвечающий за биты регистра PORTxn.

Как видно из схемы, значения выходов “Q“ этих D-триггеров и определяют поведение пина. Например, если пин сконфигурирован для работы на ввод (DDRxn = 0) и в регистр PORTxn записана логическая единица, выходы D-триггеров, подключённые к элементу «И», образуют сигнал, который с помощью полевого транзистора подключит подтягивающий резистор к напряжению питания.

Регистр PINxn обходит стороной всю эту логику и просто позволяет считать непосредственное значение пина.

Приведём таблицу значений регистров и режимов работы порта.

DDxn	**PORTxn**	**PUD**	**Режим**	**Подтягивающий резистор**	**Комментарий**
0	0	×	ввод	нет	Высокоимпедансное (неопределённое) состояние
0	1	0	ввод	да	Pxn будет источником тока
0	1	1	ввод	нет	Высокоимпедансное (неопределённое) состояние
1	0	×	вывод	нет	Логический «0» (низкий уровень, земля)
1	1	×	вывод	нет	Логическая «1» (высокий уровень, источник напряжения)

PUD — это специальный бит в регистре SFIOR, который позволяет отключить подтягивающие резисторы сразу для всех пинов всех портов.

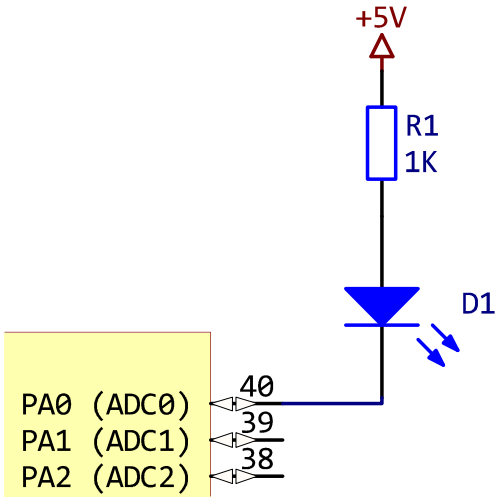
Подтягивающий резистор

При работе пина на вывод к нему можно подключить встроенный подтягивающий резистор, записав «1» в соответствующий бит регистра PORTx. Через этот резистор на пин подаётся напряжение питания, вследствие чего пин становится источником тока и по умолчанию имеет значение «1» (высокий уровень). Подтягивающие резисторы используются при работе с кнопками, когда один вывод кнопки подключается к пину, а другой — к земле. При разомкнутой кнопке пин имеет всё то же значение «1», а при замкнутой ток уходит в землю и пин принимает значение «0».

Подключение светодиодов и кнопок

В этом разделе мы подробно рассмотрим способы подключения светодиодов и кнопок к микроконтроллеру, а также разберём примеры кода, позволяющего их использовать.

Начнём с самого простого: подключения светодиода.

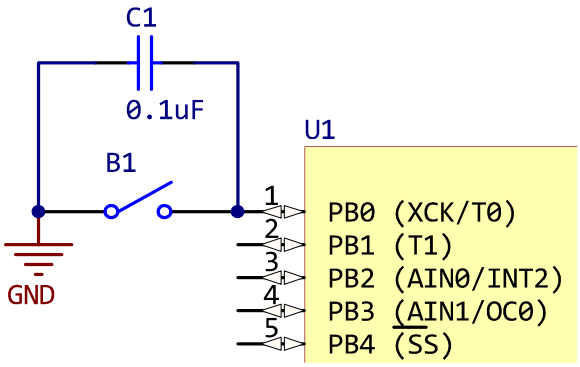


Как видно из схемы, к источнику напряжения подключается резистор, который служит для ограничения тока, протекающего через диод. Сам светодиод подключается анодом к резистору, а катодом — к пину микроконтроллера. Очевидно, что для включения светодиода нам нужно создать положительную разность потенциалов между анодом и катодом, а для выключения — уровнять напряжения, приложенные к ним. Вспомнив табличку режимов работы порта, делаем вывод, что для работы с диодом мы должны сконфигурировать соответствующий пин для работы на вывод. После этого мы можем записать «0» в соответствующий бит регистра PORTx и, создав таким образом на пине низкий уровень напряжения, включить светодиод. Чтобы выключить светодиод, достаточно записать в нужный бит «1». Это своеобразная инвертированная логика, к которой нужно привыкнуть.

В коде это будет выглядеть так:

```
DDRA |= (1 << PA0);// Set pin 0 of port A to output
PORTA |= (1 << PA0);      // Turn off LED
PORTA &= ~(1 << PA0);     // Turn on LED
```

В подключении кнопки к микроконтроллеру тоже нет ничего сложного.



На схеме показано, как кнопка одним концом подключается к пину контроллера, а другим — к земле. Кроме этого, параллельно кнопке подключен конденсатор для предотвращения «дребезга» при нажатии кнопки.

Очевидно, что для работы с кнопкой пин должен быть сконфигурирован для работы на ввод. Кроме того, в данной конфигурации нужно включить подтягивающий резистор. В этом случае, значение бита в регистре PINx будет равно «1» при разомкнутой кнопке и «0» — при замкнутой. Всё та же инвертированная логика, к которой нужно привыкнуть.

В коде работа с кнопкой выглядит так:

```
DDRB &= ~(1 << PB0);      // Set pin 0 of port B to input
PORTB |= (1 << PB0);      // Turn on pull-up resistor

if (PINB & (1 << PB0))
      // Button is depressed
else
      // Button is pressed
```

Если пин используется для работы с кнопкой на протяжении всей программы, первую инструкцию (задание режима работы на ввод) можно опустить, так как по умолчанию все пины сконфигурированы на ввод.

Среда разработки

Для написания и отладки программ под микроконтроллеры линейки AVR используется среда разработки под названием AVR Studio. Она позволяет создавать проекты, состоящие из множества файлов с исходным кодом, компилировать их, и загружать в микроконтроллер при помощи программатора. Также среда предоставляет возможности для отладки программ, которые выполняются непосредственно в контроллере.

Установка среды разработки

Скачать AVR Studio можно с официального сайта Atmel по следующей ссылке: http://atmel.com/dyn/products/tools_card_v2.asp?tool_id=2725 (специалисты компании Atmel, судя по всему, не слышали про SEO, поэтому ссылка может измениться). Установка программы не составляет никаких трудностей: достаточно соглашаться со всем, что предложит инсталлятор.

Однако на этом установка не заканчивается. Дело в том, что AVR Studio предоставляет «из коробки» только средства для разработки на ассемблере. Мы же будем писать программы на C, поэтому нам нужно дополнительно установить тулчейн[1] под названием WinAVR. Найти последнюю версию можно на странице проекта в SourceForge: http://sourceforge.net/projects/winavr/files/. В установке WinAVR также нет ничего сложного. Кроме того, WinAVR автоматически интегрируется с AVR Studio, так что никаких дополнительных настроек делать не нужно.

Пара слов об альтернативной среде

AVR Studio предоставляет довольно скудные возможности в плане редактирования кода и удобства использования. Она не содержит практически никаких инструментов, присущих современным средам разработки: в ней нет рефакторинга, адекватной подсветки синтаксиса, подсказок о параметрах функций и других необязательных, но очень приятных мелочей.

К счастью, мы не прикованы к стандартному инструментарию, предоставляемому компанией Atmel. В качестве альтернативной среды может использоваться платформа Eclipse с установленными плагинами для разработки на C и специальными средствами для программирования под микроконтроллеры AVR. Безусловно, разобраться с этой средой сложнее, чем с AVR Studio. Однако если вы поймёте, что стандартная среда вас по каким-то причинам не устраивает, всегда есть возможность перейти на Eclipse.

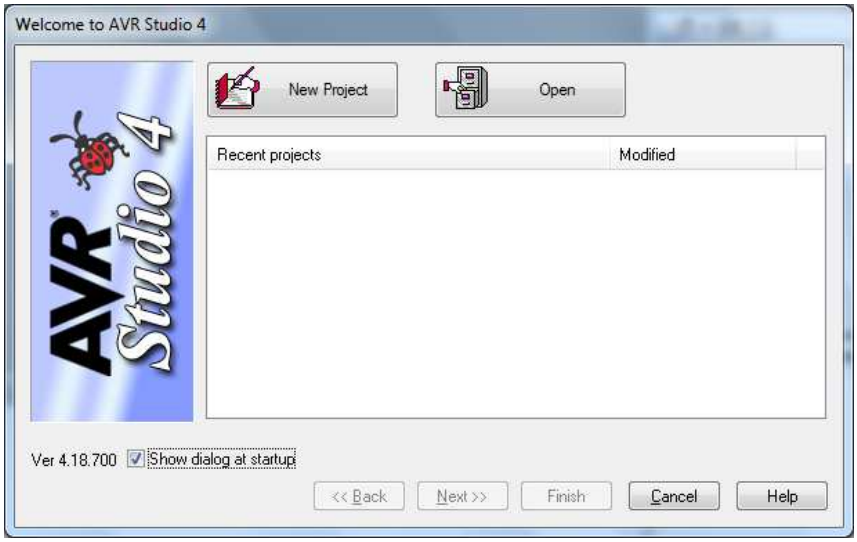
В этом пособии мы будем рассматривать только AVR Studio, так как данная среда является стандартом, и мы учимся в университете, а не в кружке «Умелые руки». Но если вам интересно попробовать альтернативу, вы можете скачать настроенную платформу Eclipse по ссылке: [ ] и попробовать разобраться с ней самостоятельно. Не имеет значения, в какой среде вы работаете; главное, чтобы работала программа.

---

[1] Тулчейн (от англ. toolchain) — набор из нескольких инструментов, необходимых для совершения нескольких действий, связанных общей целью. Например, сборка программы состоит из операций компиляции и линковки, для каждой из которых существует отдельный инструмент. Термин «тулчейн», как и многие другие, используемые в этом пособии, является сленговым. Эти термины применяются сознательно с тем, чтобы донести суть материала в как можно более простой форме, не перегружая текст надуманными и зачастую слишком усложнёнными «академическими» терминами.

Создание проекта

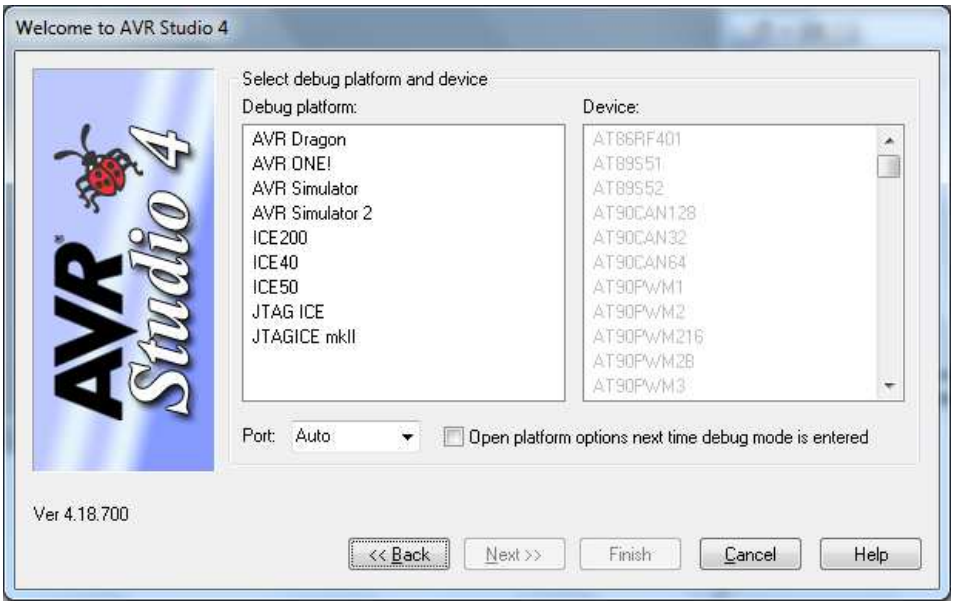
При запуске AVR Studio открывается окно, в котором можно создать новый проект, открыть существующий проект с диска или открыть один из недавних проектов.



Для создания нового проекта нажмём кнопку “New Project”.

На первом экране мастера выберем тулчейн “AVR GCC”, с помощью которого мы и будем компилировать программы на языке С. Также на этом экране указывается имя проекта и папка, в которую его нужно сохранить. В поле Location удобно указать общую папку для всех проектов AVR и отметить пункт “Create folder”. В этом случае в общей папке будет автоматически создаваться подпапка для каждого нового проекта.

Нажав нопку “Next”, мы переходим на экран выбора платформы.



Здесь мы выбираем платформу, с помощью которой будет проводиться отладка программы.

Существует два основных способа отладки: на «живом» контроллере с помощью аппаратного отладчика, или без участия каких-либо внешних устройств в программном симуляторе микроконтроллера. У каждого из способов есть свои плюсы и минусы: так, отладка в симуляторе происходит гораздо быстрее (за счёт того, что среде не нужно обмениваться данными с микроконтроллером), тогда как внутрисхемная отладка позволяет проследить работу контроллера в реальной системе.

Для отладки в симуляторе в списке “Debug platform” нужно выбрать “AVR Simulator” или “AVR Simulator 2”. Эти симуляторы отличаются списком поддерживаемых контроллеров.

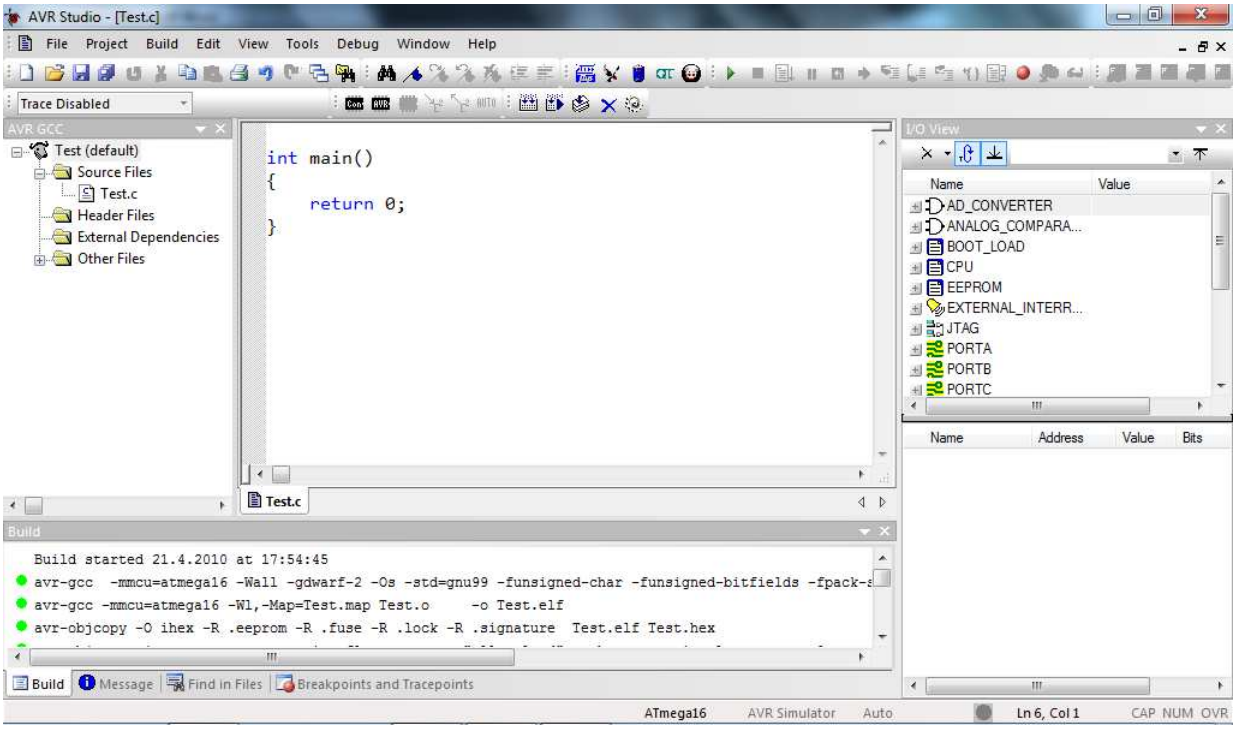
Для внутрисхемной отладки нужно выбрать ту платформу, которую вы используете. В этом курсе мы будем применять в качестве программатора и отладчика клон JTAG ICE, поэтому из списка нужно выбирать соответствующую опцию. Кроме того, нужно отметить пункт “Open platform options…”, который позволит настроить отладчик при первом запуске.

После выбора платформы нужно выбрать модель микроконтроллера из списка “Device”. Серым выделены контроллеры, которые не поддерживаются выбранной платформой. Мы используем контроллер ATmega16.

После всех этих несложных действий нужно нажать кнопку “Finish” для завершения работы мастера. Ваш проект будет создан, и вы увидите главное окно AVR Studio во всей красе.

Главное окно

Посмотрим, что собой представляет главное окно AVR Studio, в котором и будет происходить вся работа:



В самом центре расположены окна редактирования программы. Можно открыть сразу несколько файлов с исходным кодом и перемещаться между ними с помощью закладок.

Слева находится список файлов, из которых состоит проект. В него включаются файлы с исходным кодом (Source files) и заголовочные файлы (Header files). В External Dependencies будут перечислены все внешние файлы с исходным кодом, которые используются в проекте, но не являются его частью (например, стандартные заголовки, идущие в комплекте avr-libc).

Справа можно увидеть панель “I/O View”. Она используется во время отладки для получения информации о текущем состоянии контроллера. В верхней части панели представлен список периферии выбранного контроллера. Если выбрать какой-нибудь пункт, в нижней части панели отобразятся регистры, относящиеся к данному устройству и их текущие значения. Мы ещё вернёмся к этой панели, когда будем отлаживать программы.

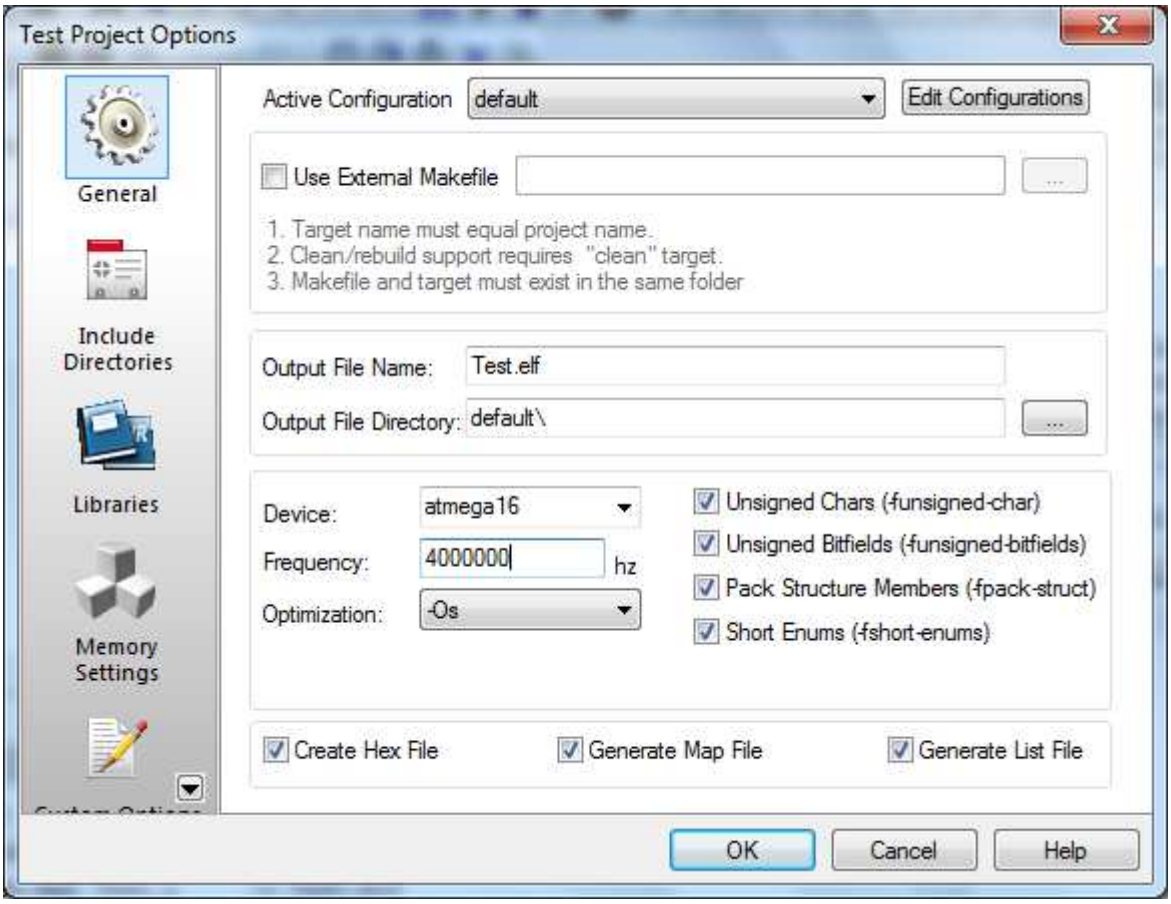
И, наконец, в самом низу находится несколько информационных панелей, по которым можно перемещаться с помощью закладок. Наиболее интересными являются панель “Build”, в которую выводится информация о процессе компиляции и панель “Breakpoints and Tracepoints”, из которой можно управлять брейкпоинтами[1].

---

[1] Брейкпоинт (от англ. breakpoint) — так же известен как точка останова. Место в программе, на котором отладчик прерывает исполнение программы. Подробнее — в разделе про отладку программ.

Настройки проекта

Каждый проект имеет некоторые настройки, которые влияют на процесс компиляции и на программу, получаемую на выходе. Окно настроек открывается с помощью пункта “Configuration options” меню “Project”.



Все настройки, которые нам придётся изменять, сгруппированы на вкладке “General” (что позволяет сэкономить кучу места на скриншотах). Наибольший интерес для нас представляют следующие опции:

- Device — модель микроконтроллера, которую мы используем в проекте.
- Frequency — тактовая частота микроконтроллера в герцах. Как уже было сказано в обзоре стенда, мы используем внешний кварц на 4 Мгц, так что в этом поле нужно установить значение 4000000.
- Unsigned chars — если эта опция включена, то переменные типа *char*, объявленные без модификаторов, будут считаться объявленными с модификатором *unsigned*. Эту опцию иногда бывает полезно отключать (если вы знаете, что делаете).
- Optimization — очень важная опция. Во время сборки выходного файла компилятор оптимизирует исходный код, который вы написали. Данная опция контролирует стратегию, которую он применяет для оптимизации.
    - –O0 — оптимизация не проводится, код компилируется так, как вы его написали.

- –O1, –O2 и –O3 — оптимизация для скорости. При выборе этих опций компилятор будет стараться сделать код максимально быстрым, меньше принимая во внимание его размер. Например, он может решить, что в каком-то из случаев умножение будет выгоднее заменить многократным сложением.
- –Os — оптимизация для минимального размера кода. При выборе этой опции компилятор постарается сделать выходной файл как можно меньшего размера, даже если для этого придётся пожертвовать производительностью.

Важно запомнить, что при отладке программы нужно полностью отключать оптимизацию (выставлять режим -O0). Если вы этого не сделаете, то получите совершенно непредсказуемые результаты, так как отладчик ожидает увидеть код, который написали вы, а компилятор выдаёт код с изменениями, которые кажутся ему оптимальными. В некоторых случаях компилятор выбрасывает целые блоки кода, и вы будете долго удивляться, почему отладчик не останавливается на ваших брейкпоинтах.

Если вы уже всё отладили и компилируете финальную версию кода, бывает полезно включить какую-нибудь оптимизацию. Какую именно — решать вам в зависимости от контекста. Если у вас маленькая программа, и в памяти контроллера остаётся много места, можно включить оптимизацию по скорости. Если же вы еле влезаете в отведённые 16 килобайт, и вам дорог каждый байт, вам определённо захочется оптимизировать размер выходного файла. Однако учтите, что компилятор иногда так оптимизирует код, что он перестаёт работать. Справедливости ради стоит заметить, что такое происходит крайне редко и преимущественно в плохо написанном коде. Если вдруг ваша программа, работавшая как часы при отладке, стала странно вести себя после оптимизации, стоит задуматься о том, насколько хорошо вы написали этот код. Или же просто отключить оптимизацию, что зачастую бывает гораздо проще.

Написание программы

В следующих нескольких разделах мы напишем простенькую программу и рассмотрим на её примере полный цикл разработки: компиляцию, прошивку и отладку.

Начнём с написания кода программы. Допустим, мы хотим сделать «бегущий огонёк» с помощью модуля светодиодов. В начале выполнения программы будет гореть первый светодиод, через какое-то время будет загораться второй и так далее. При включении следующего диода предыдущий должен гаснуть, а при достижении последнего диода должен снова загораться первый.

Итак, создадим новый проект под названием LEDTest. Среда автоматически создаст файл LEDTest.c и откроет его в редакторе.

Первым делом нам нужно подключить заголовочные файлы, позволяющие работать с портами и организовывать задержку. Для этого в начале LEDTest.c добавим директивы *#include*:

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

В первом заголовочном файле находятся макросы для обозначения регистров (чтобы можно было обращаться к регистрам по именам, а не только по адресам). Во втором файле находятся прототипы функций, реализующих задержку.

Лирическое, но от того не менее важное отступление: *Описание всех заголовочных файлов вы можете найти в документации по avr-libc. Для этого в меню "Help" AVR Studio выберите пункт "avr-libc Reference Manual". На открывшейся странице перейдите в раздел "Reference manual", в котором и находится список всех заголовочных файлов. Если нажать на любой из заголовков, откроется список всех функций и макросов, которые в нём содержатся. Документация к библиотеке является одним из основных источников информации, наряду с документацией на микроконтроллер и поисковиком Google.*

Итак, нужные функции и макросы у нас есть, можно писать код. Допустим, что мы подключили модуль светодиодов на порт А. Чтобы управлять светодиодами, мы должны установить порт А в режим вывода. Это можно сделать с помощью инструкции:

```
DDRA = 0xff;
```

Мы присвоили регистру DDRA значение 0xff (или десятичное 255) установив, таким образом, все его биты в "1".

Теперь нам нужно подумать, как реализовать нашу задачу — «бегущий огонёк». Так как заказчик не указал в техническом задании ничего, что касалось бы времени выполнения программы или условий её остановки, предположим, что программа должна выполняться бесконечно. Раз так, мы можем написать бесконечный цикл с помощью конструкции *while (1)*. Но что мы будем делать в этом цикле? Применим системный подход.

- В каждый момент времени будет гореть только один светодиод
- Светодиоды будут зажигаться по порядку
- За последним светодиодом идёт первый
- Между переключениями светодиодов есть фиксированная пауза

Первые три условия дают нам очевидное решение: в начале программы мы присваиваем регистру PORTA значение *254*, тем самым зажигая первый светодиод (вспоминаем инвертированную логику управления светодиодами и тот факт, что *254* в двоичной системе будет выглядеть как *11111110*). После этого, в каждой итерации цикла мы делаем побитовый сдвиг текущего значения регистра с переносом старшего бита. Это означает, что последний бит в значении будет не затираться, как в обычном побитовом сдвиге, а переноситься на место первого бита. И, наконец, четвёртое условие предписывает нам генерировать задержку в начале каждой итерации цикла. Таким образом, весь код в файле LCDTest.c будет выглядеть так:

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

int main()
{
     DDRA = 0xff;
     PORTA = 254;
     while (1) {
           _delay_ms(500);
           PORTA = (PORTA << 1) | ((PORTA & 0x80) >> 7);
     }

}
```

Рассмотрим подробнее реализацию сдвига с переносом. Инструкция

```
PORTA = (PORTA << 1) | ((PORTA & 0x80) >> 7);
```

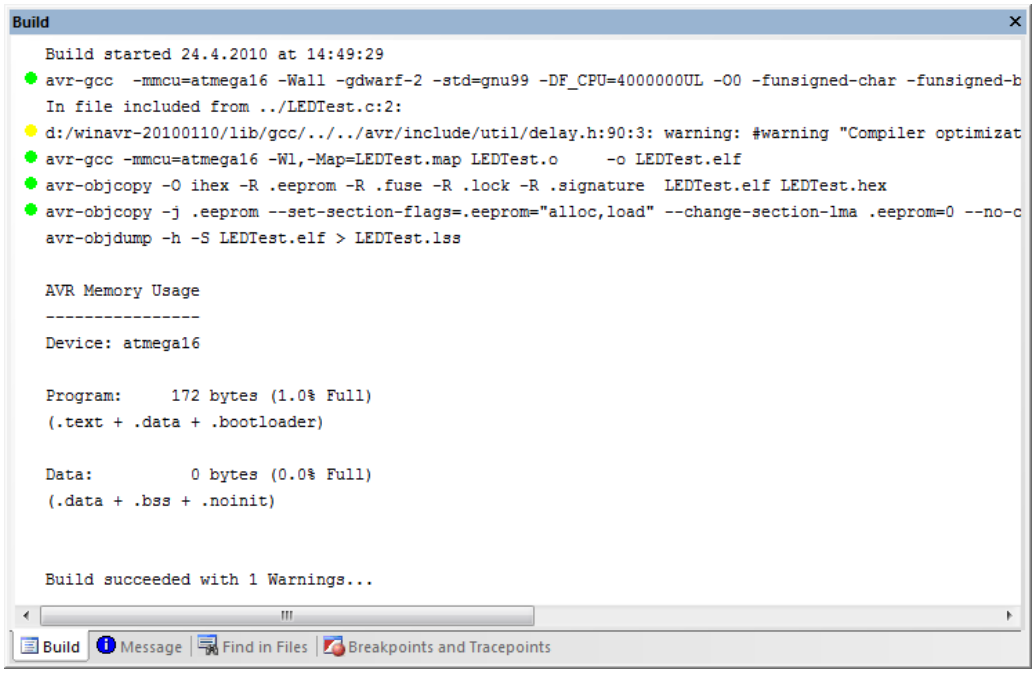
состоит из двух основных частей. Часть *(PORTA << 1)* — это обычный побитовый сдвиг. Если бы мы использовали только его, значение последнего бита было бы затёрто, а первый бит был бы установлен в «0». Однако здесь в дело вступает вторая часть инструкции, *((PORTA & 0x80) >> 7)*. Эта часть считывает текущее значение последнего бита и делает побитовый сдвиг вправо, перенося его на позицию первого бита. После этого, обе части объединяются с помощью операции «ИЛИ».

Функция *_delay_ms* реализует задержку на указанное количество миллисекунд. В данном случае мы сделали задержку в полсекунды.

Всё остальное вполне очевидно, особенно если освежить память прочтением краткого описания по С.

Компиляция программы

Чтобы скомпилировать программу и получить файл, готовый для загрузки в микроконтроллер, достаточно нажать клавишу F7 (или в меню “Build” выбрать пункт “Build”). В нижней панели на закладке “Build” будет выведен отчёт о компиляции. Развёрнутый отчёт для нашей программы показан на рисунке:



В последней строке указан результат сборки: успешно ли она прошла, а так же количество ошибок и предупреждений. В нашем случае было выдано одно предупреждение об отключенной оптимизации. Дело в том, что функции *_delay** работают правильно только при включённой оптимизации, тогда как мы отключили их в целях отладки.

Выше в отчёте располагается информация о том, сколько места во флэш-памяти занял код программы, а ещё выше можно найти все выданные предупреждения и ошибки. Предупреждения отмечены жёлтым кружком, а ошибки — красным. Предупреждения не влияют на успешность сборки программы, однако, лучше всегда стремиться к коду, который не выдаёт предупреждений вообще. Таким образом можно обезопасить себя от неожиданных и трудноуловимых багов.

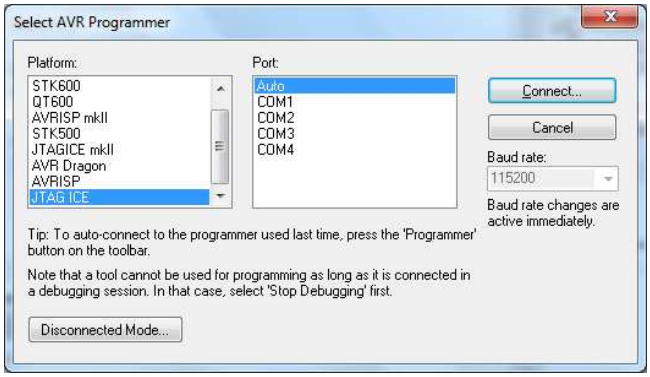
После сборки программы в папке “default” вашего проекта будет лежать файл с именем *{имя_проекта}.hex*. Именно его нужно загружать в микроконтроллер.

Загрузка программы в микроконтроллер

Для загрузки программы используются две вещи: аппаратный программатор и программа-загрузчик. В нашем курсе мы используем совмещённый программатор и отладчик JTAG ICE (точнее, его клон), в то время как AVR Studio содержит встроенную программу для загрузки прошивок.

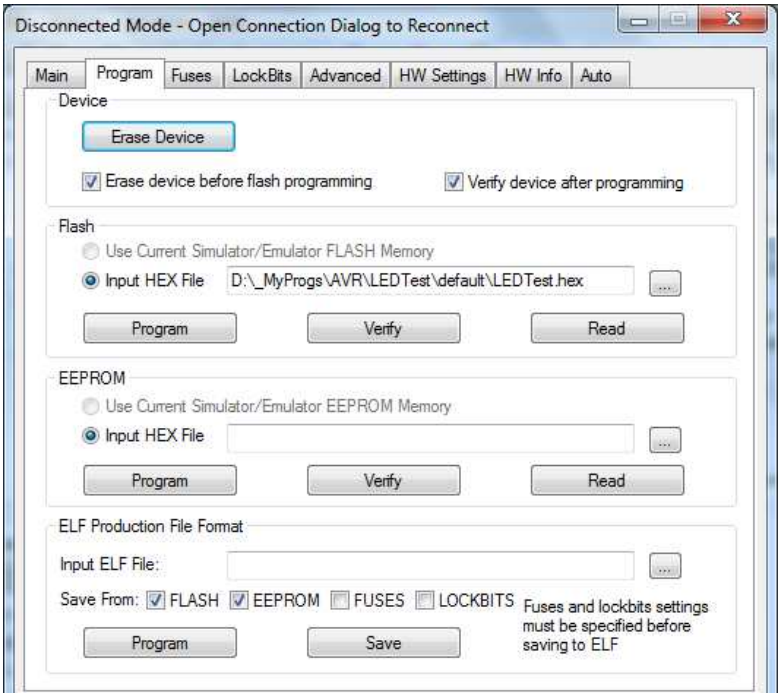
Чтобы начать загрузку программы, нужно подключить программатор компьютеру через USB и к основному модулю с помощью специального разъёма. Далее надо выбрать модель программатора в AVR Studio. Для нажимаем кнопку “Display the ‘Connect’ dialog” на панели инструментов: 

В открывшемся окне выбираем “JTAG ICE” и нажимаем кнопку “Connect”.



После того, как вы один раз выберете программатор и успешно к нему подключитесь, можно пропустить окно выбора устройства нажатием на кнопку “Connect to a selected programmer” на панели инструментов.

Если подключение к программатору прошло успешно, откроется такое окно (не совсем такое, скриншот был сделан в disconnected mode без программатора):



В этом окне есть много интересных опций, но нас в данный момент интересует только группа “Flash”. Здесь вы можете указать путь к скомпилированному файлу с прошивкой, а также загрузить её в контроллер с помощью кнопки “Program”.

Внимание! *Среда не проставляет путь к прошивке в зависимости от открытого проекта! Если вы стали работать над другим проектом, убедитесь, что в поле “Input HEX File” прописан путь к нужной прошивке. Если об этом забыть, можно случайно загрузить в контроллер прошивку от предыдущего проекта, после чего потратить много времени, удивляясь странному поведению контроллера.*

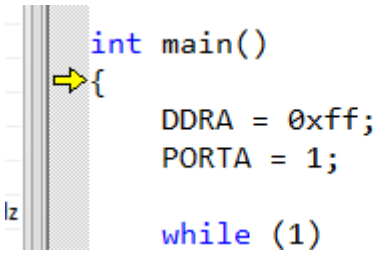
После указания пути и нажатия кнопки “Program” прошивка будет загружена в контроллер и заново прочитана, чтобы удостовериться в правильной загрузке. После окончания процесса загрузки контроллер будет перезапущен, и ваша программу сразу начнёт выполняться.

Отладка

Отладка — важный этап в процессе написания программы. Очень редко бывает так, что программа работает правильно после первой же компиляции (если это не “Hello world”, хотя, некоторые и там умудряются делать ошибки). Во время отладки можно контролировать ход выполнения программы, просматривать значения переменных и регистров на каждом шаге и даже, если это критично для вашей программы, измерять количество тактов, за которое выполняется тот или иной блок.

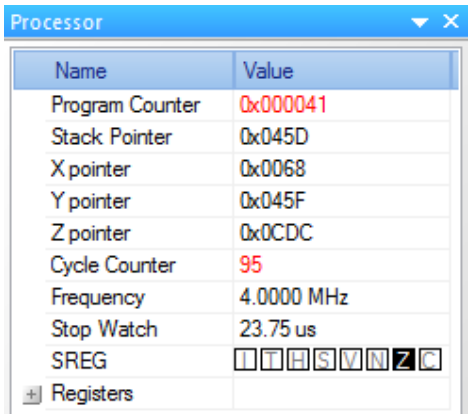
Чтобы запустить отладку в AVR Studio, достаточно нажать кнопку “Start Debugging” на панели инструментов:  . Если вы используете внутрисхемную отладку, убедитесь, что программатор подключён к контроллеру и в схему подано питание. Перед началом отладки ваша программа будет автоматически загружена в микроконтроллер. Если же вы отлаживаете в симуляторе, никаких дополнительных действий не требуется.

После инициализации и запуска отладчик остановится на первой инструкции вашей программы. Инструкция, которая будет выполнена на следующем шаге, обозначена жёлтой стрелкой слева от редактора кода.



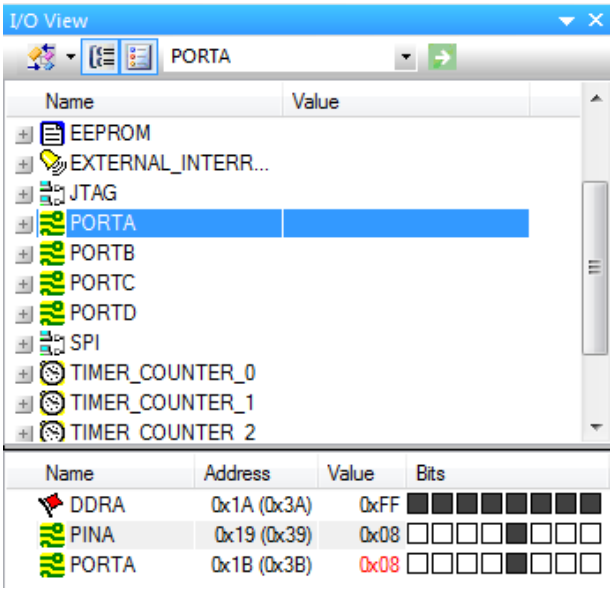
Заметьте, что в начале выполнения программы стрелка указывает не на инструкцию *DDRA = 0xff*, а на открывающую скобку блока функции *main*. Дело в том, что при входе в функцию контроллер производит дополнительные действия, такие как извлечение аргументов функции из стека.

Слева от редактора кода можно увидеть новое окно, которое автоматически открывается в режиме отладки: “Processor”.



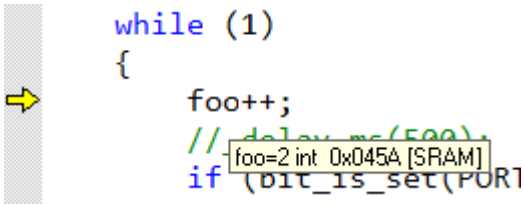
В этом окне представлена информация о текущем состоянии процессора. В него входит адрес следующей инструкции, указатели стека, регистры общего назначения, счётчик циклов и тактовая частота.

В то время, пока работа программы приостановлена, вы можете изучить содержимое специальных регистров в окне “I/O View”.

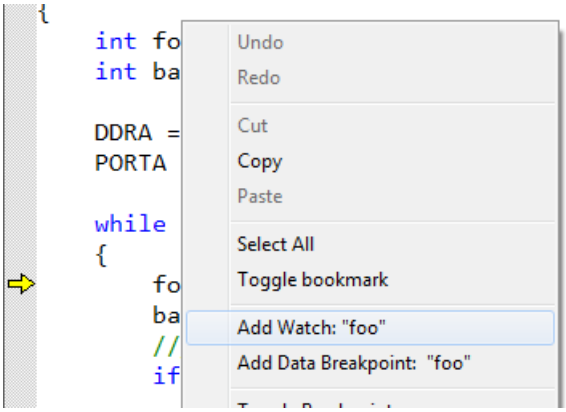


Достаточно выбрать интересующий вас объект в верхней части окна, и в нижней части отобразятся нужные регистры и их значения. В данном случае мы просматриваем значения регистров порта А. Биты, имеющие значение «1», закрашены тёмно-серым.

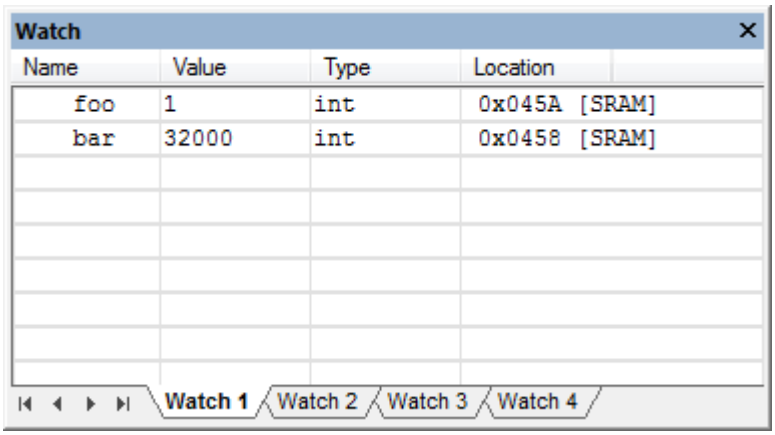
Если вы используете свои переменные, то всегда можете узнать их значение, просто подведя мышь к любому месту, в котором эта переменная используется (конечно, это работает только когда программа приостановлена).



Если вы хотите наблюдать за несколькими переменными сразу, вы можете добавить их в окно наблюдения. Для этого нажмите правой кнопкой на переменную и выберите “Add watch”.



После этого появится окно “Watch”, в котором вы сможете следить за выбранными переменными.



Теперь рассмотрим способы управления выполнением программы. Первый, и самый простой способ — это пошаговое выполнение. При нажатии на клавишу F10 отладчик выполняет одну инструкцию и снова приостанавливает выполнение программы. Этого же эффекта можно добиться, нажав кнопку “Step over”: . Если следующая инструкция является вызовом функции, то при нажатии F10 эта функция будет выполнена полностью, а программа остановлена на следующей за этой функцией инструкции. Чтобы войти в функцию и так же пошагово её отладить, нужно использовать команду “Step into”: (F11 на клавиатуре). После этой команды отладчик войдёт в функцию и остановится перед выполнением первой её инструкции. Если применить команду “Step into” на инструкции, которая не является вызовом функции, то она сработает как “Step over”. Для возврата в вызывающую функцию используется команда “Step out”: (Shift + F11). При её использовании текущая функция выполняется до конца без остановок, после чего управление возвращается вызывающей функции и программа останавливается на следующей инструкции.

При выборе команды “Run to cursor” ( ) программа будет выполняться без остановки до того места, на которое в данный момент установлен курсор ввода.

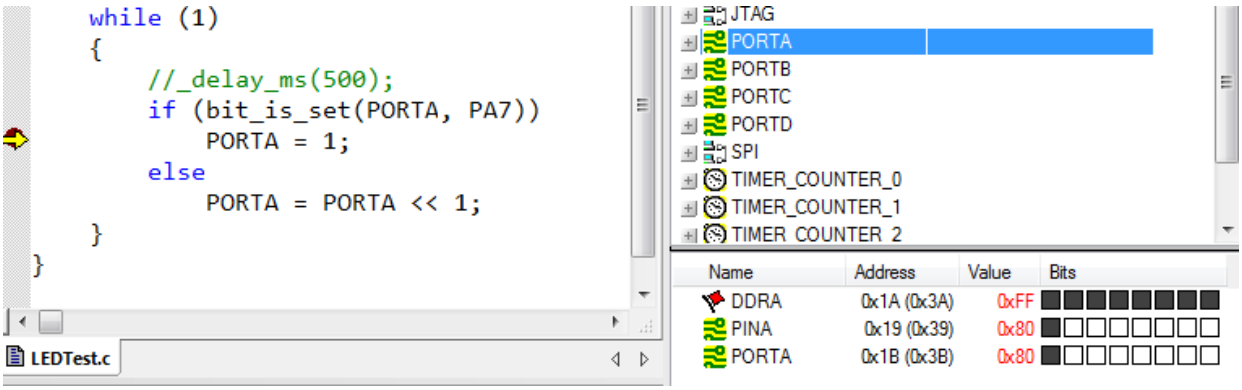
С помощью команды “Autostep” ( ) можно запустить автоматическое выполнение программы с небольшой остановкой на каждом шаге. Аналогичного эффекта можно добиться, положив что-нибудь тяжёлое на клавишу F10.

И, наконец, при помощи команды “Run” (, F5) вы можете запустить непрерывное выполнение программы. В любой момент можно нажать кнопку паузы, чтобы программа остановилась на текущей инструкции.

Теперь рассмотрим, пожалуй, самый эффективный метод отладки: точки останова (или брейкпоинты). С их помощью вы можете точно определить места, в которых должна останавливаться ваша программа и не тратить время на пошаговое выполнение блоков, работа которых уже проверена. Брейкпоинты имеют приоритет над любыми другими командами. Это означает, что ваша программа будет останавливаться на них всегда, каким бы способом вы её не отлаживали. Чтобы установить брейкпоинт на определённой строке, достаточно поместить туда курсор и нажать F9. Красный кружок слева от строки известит вас о том, что программа теперь всегда будет останавливаться на этой строке. Допустим, мы хотим остановиться на той инструкции в нашей программе, которая присваивает порту А значение «1» при переключении с последнего светодиода.

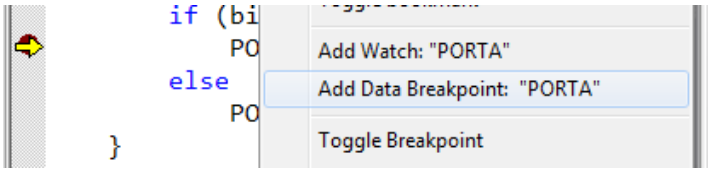
```
#include <avr/io.h>
#include <util/delay.h>

int main()
{
    DDRA = 0xff;
    PORTA = 1;

    while (1)
    {
        //_delay_ms(500);
        if (bit_is_set(PORTA, PA7))
            PORTA = 1;
        else
            PORTA = PORTA << 1;
    }
}
```

Теперь запустим отладку и нажмём F5, чтобы программа начала выполняться без остановок. Как видите, отладчик автоматически выполнял все инструкции, пока не наткнулся на ту, где установлен брейкпоинт. Если посмотреть в этот момент в окно “I/O View”, мы увидим, что в данный момент включен последний бит порта А, как и должно быть по логике программы.



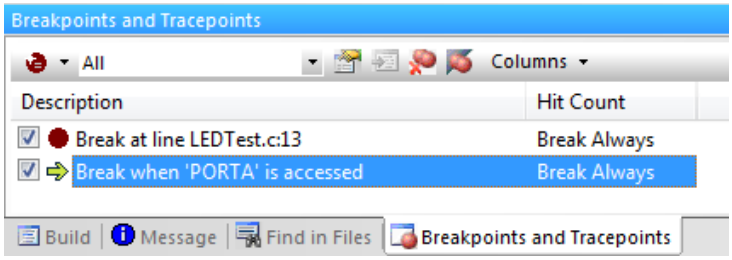
Чтобы продолжить выполнение программы, достаточно снова нажать F5 или использовать любую другую команду из рассмотренных ранее.

Кроме брейкпоинтов, реагирующих на выполнение программы, вы можете устанавливать точки останова, следящие за доступом к памяти. Например, мы можем установить брейкпоинт на доступ к порту A.

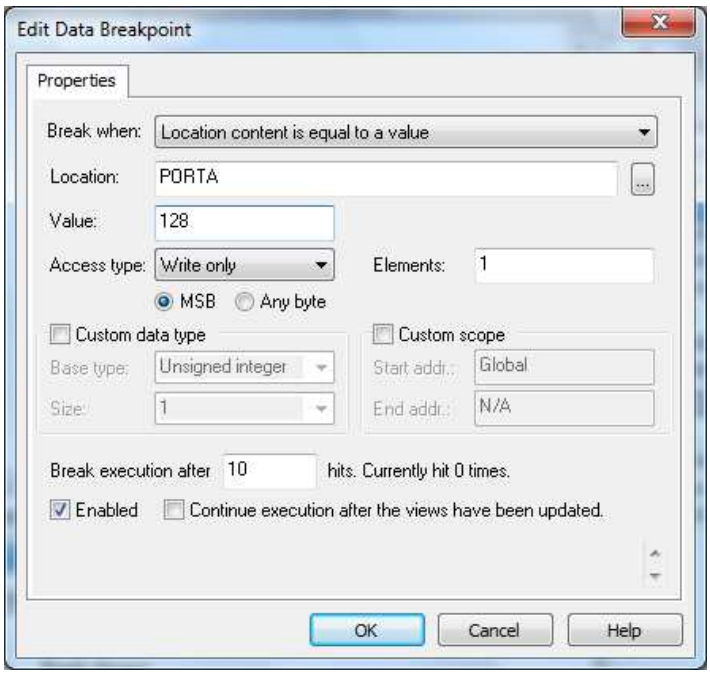


Теперь программа будет останавливаться при каждом обращении к порту A (неважно, что это будет за операция: чтение, или запись).

Для управления брейкпоинтами можно использовать панель “Breakpoints and Tacepoints”. В ней перечислены все установленные брейкпоинты и условия их срабатывания.



Условие срабатывания можно изменить, щёлкнув правой кнопкой по брейкпоинту и выбрав “Properties”. Так, например, выглядят свойства data breakpoint:



В данном случае мы изменили свойства так, что остановка на брейкпоинте будет выполнена только если текущее значение порта A будет равно 128 (установлен последний бит), в следующей инструкции будет производиться запись в порт и все эти условия были выполнены уже не менее 10 раз (то есть, программа могла бы остановиться в этом месте уже 9 раз, но, по нашему условию остановиться только на 10).

Примеры

В разделе, посвящённому среде разработки, мы уже разобрали одну простейшую задачу на работу со светодиодами. Теперь посмотрим, как можно работать с кнопками.

Обработка нажатия кнопок

Допустим, вы подключили модуль кнопок на пины PD0-PD3. Как уже было сказано выше, состояние кнопки можно проверить, прочитав соответствующий бит нужного порта. Напишем код, проверяющий нажатие кнопки, с помощью функции *bit_is_clear()*:

```
        if (bit_is_clear(PIND, PD0))
             // Кнопка нажата
        else

             // Кнопка не нажата
```

Существует несколько библиотечных функций, облегчающих работу с отдельными битами. Среди них функции *bit_is_set()* и *bit_is_clear()*. *bit_is_set()* проверяет, имеет ли указанный бит указанного регистра значение «1». Эта функция аналогична записи:

```
        if (PORTA & (1 << PA7))
```

Функция *bit_is_clear()* аналогична *bit_is_set()*, с той лишь разницей, что возвращает *true* когда указанный бит имеет значение «0».

Модифицируем прошлый пример с «бегущим огоньком» так, чтобы светодиоды переключались не через интервал времени, а только при нажатии кнопки. Предположим, что модуль кнопок подключён к пинам PD0-PD3, а модуль светодиодов — к порту A.

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

int main()
{
       DDRA = 0xff; // Порт A на вывод
       PORTA = 1;   // Установить первый бит порта A в 1
       PORTD = 1;   // Включить подтягивающий резистор на PIND0

       char lastState = 1;// Переменная для хранения состояния кнопки
       while (1)    // Бесконечный цикл
       {
              char currentState = bit_is_clear(PIND, PD0);
              if (currentState && !lastState)
              {  // Если на прошлом такте кнопка была не нажата, а на этом уже нажата
                     PORTA = (PORTA << 1) | ((PORTA & 0x80) >> 7);
              }
              lastState = currentState;
       }
}
```

Задания для лабораторной работы

Простые

1. При нажатии и удержании одной из кнопок горят светодиоды 1-4; при отпущенной кнопке горят светодиоды 5-8.
2. Однократное нажатие кнопки меняет местами горящие и не горящие группы диодов местами. При нажатии кнопки горят диоды 1-4; при повторном нажатии — 5-8.
3. Управление положением горящего диода с помощью кнопок. Реализовать нопку «вверх» и кнопку «вниз». Крайние положения обрабатывать особо.
4. Реализовать «Бегущий огонёк». При достижении последнего (первого) светодиода движение начинается в другую сторону.

Средние

1. «Бегущий огонёк» с тремя светодиодами и заходом за край гребёнки:

   ×××××××
   ¤×××××××
   ¤¤××××××
   ¤¤¤×××××
   ×¤¤¤××××
   ××¤¤¤×××
   ×××¤¤¤××
   ××××¤¤¤×
   ×××××¤¤¤
   ××××××¤¤

2. «Тетрис». Тот же бегущий огонёк, но при достижении другого конца платы, последний незажжённый диод остаётся гореть.

   o×××
   ×o××
   ××o×
   ×××o
   o××o
   ×o×o
   ××oo

3. И снова вариация огонька. На этот раз огоньку придётся бежать сразу в две стороны, начиная с середины.

   ××o××
   ×o×o×
   o×××o

4. Задание 4, но два огонька, достигнув краёв платы, возвращаются к середине.

5. Придумать собственный закон изменения горящих и негорящих диодов, реализовать его и обосновать пользу в народном хозяйстве.
6. Реализовать «светофор»: три светодиода циклически переключаются с разной временной задержкой. При этом, есть кнопка перехода в ручной режим, в котором переключение диодов контролируется другой кнопкой.

Сложные

1. После сброса контроллер ждёт нажатий кнопок. Нажимаются n любых кнопок. После нажатия последней контроллер проигрывает получившийся паттерн на диодах.
2. То же, что и 1, но запоминается не только последовательность нажатия кнопок, но и интервал между нажатиями.
3. Контроллер запоминает нефиксированное число нажатий. Нажали кнопку 1 — начали запоминать новое положение. Кнопками 2 и 3 указывается диод (ф-ции « и »), кнопка 4 запускает это чудо на выполнение.

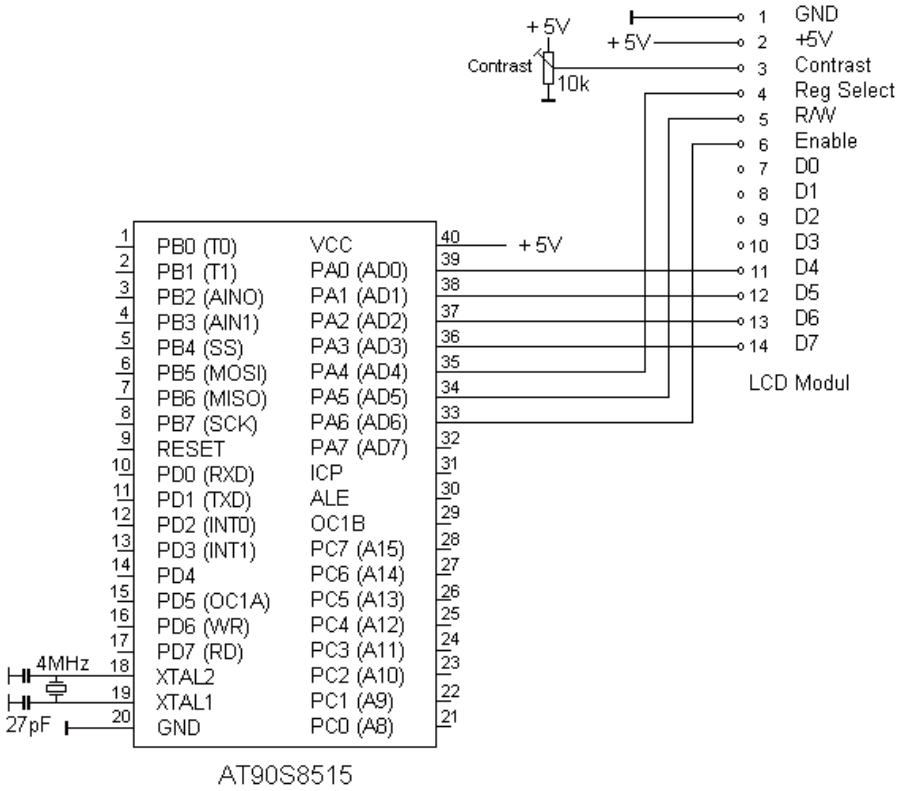
Лабораторная работа №2

В этой работе мы познакомимся с таким интересным компонентом, как LCD-дисплей. С помощью дисплеев ваши устройства могут выводить некую служебную информацию, а при наличии кнопок ещё и реализовывать меню, позволяющее настроить какие-нибудь параметры.

Большинство знакосинтезирующих дисплеев поставляются со встроенными контроллерами, которые непосредственно управляют LCD-матрицей. Всё, что остаётся сделать нам как разработчикам — это запрограммировать общение нашего контроллера с контроллером дисплея. Всю информацию о протоколе обмена данными можно почерпнуть из документации на конкретный дисплей. Однако мы не будем тратить время на написание низкоуровневых функций и в данной работе воспользуемся уже готовой библиотекой для работы с дисплеем.

Подключение дисплея к микроконтроллеру

Существует два способа подключения дисплея: по полной 8-битной шине и по 4-битной шине. Для передачи данных в этих случаях используется восемь и четыре вывода соответственно. В обоих случаях требуются три дополнительных вывода для подачи служебных сигналов на контроллер дисплея. В нашем курсе дисплей подключается по 4-битной шине, что позволяет задействовать только 7 бит одного порта. Типовая схема подключения приведена на рисунке:



В данном случае дисплей подключён к выводам порта A, однако, библиотеку можно настроить на работу с любым другим доступным портом. Выводы D4-D7 отвечают за передачу данных, в то время как с помощью выводов RS, RW и E происходит непосредственное управление контроллером дисплея. Более подробные сведения о

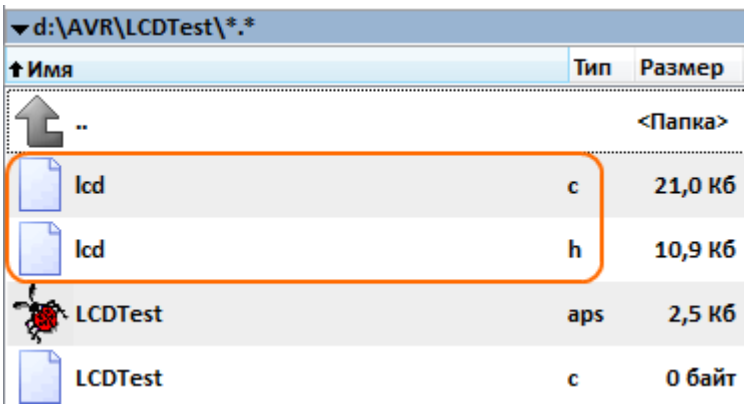
дисплее и принципах его работы не входят в минимальный уровень знаний этого курса, однако, всю информацию можно найти в документации на дисплей.

Подключение сторонней библиотеки к проекту

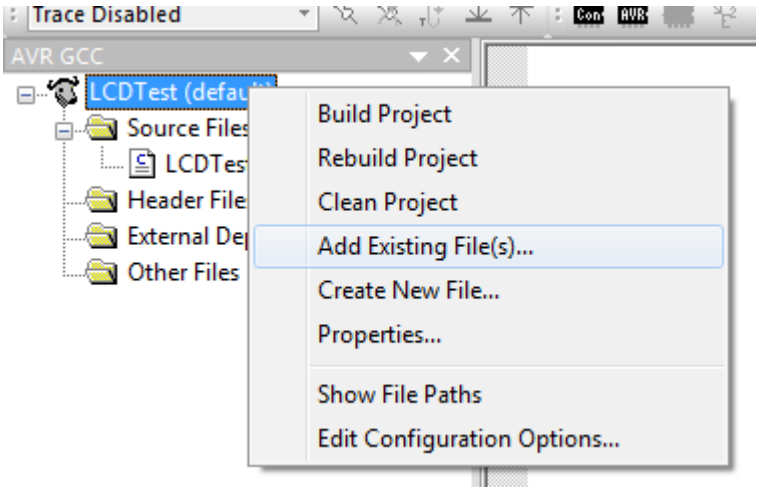
Библиотеки, написанные другими разработчиками, могут попасть к вам как в виде набора заголовков и бинарных файлов, требующих линковки, так и в виде исходного кода, который вы просто включаете в проект и компилируете вместе со своей программой. В этой лабораторной работе мы рассмотрим именно второй вариант.

Для управления LCD-дисплеем используется библиотека, написанная Peter Fleury. Подробную информацию об этой библиотеке и о способах подключения дисплеев к микроконтроллеру можно найти на домашней странице автора.

Создадим новый проект под названием LCDTest. Для подключения библиотеки нам нужно добавить в проект два файла: заголовок *lcd.h* и файл с исходным кодом *lcd.c*. Для начала скопируем эти два файла в папку проекта, которую создала для нас AVR Studio:



Теперь добавим эти файлы в сам проект, чтобы включить их в процесс компиляции. Для этого перейдём в AVR Studio и в списке файлов проекта нажмём правой кнопкой на сам проект (корневой элемент). В открывшемся меню выберем пункт “Add existing files” и выберем только что скопированные файлы.



После выполнения этих действий среда автоматически начнёт компилировать добавленные файлы с исходным кодом. Однако прежде чем использовать библиотеку, её

придётся немного настроить.

Настройка библиотеки

Параметры библиотеки, поддающиеся настройке, представлены в заголовочном файле *lcd.h* в виде констант, заданных директивами *#define*. Все параметры имеют комментарии, поэтому запутаться в них сложно. Рассмотрим основные параметры.

- `#define XTAL [freq]` — задаёт частоту, на которой работает наш микроконтроллер. По каким-то причинам автор не воспользовался стандартным макросом *F_CPU*, поэтому нам придётся вручную установить это значение в 4000000 (4 МГц).
- `#define LCD_LINES [lines]` — задаёт количество строк дисплея
- `#define LCD_DISP_LENGTH [len]` — количество символов в одной строке
- `#define LCD_PORT [PORTx]` — порт, к которому подключён дисплей

В нашем курсе дисплей обычно подключается к порту B, и в шаблонном файле *lcd.h* уже указаны правильные значения.

Описание функций библиотеки

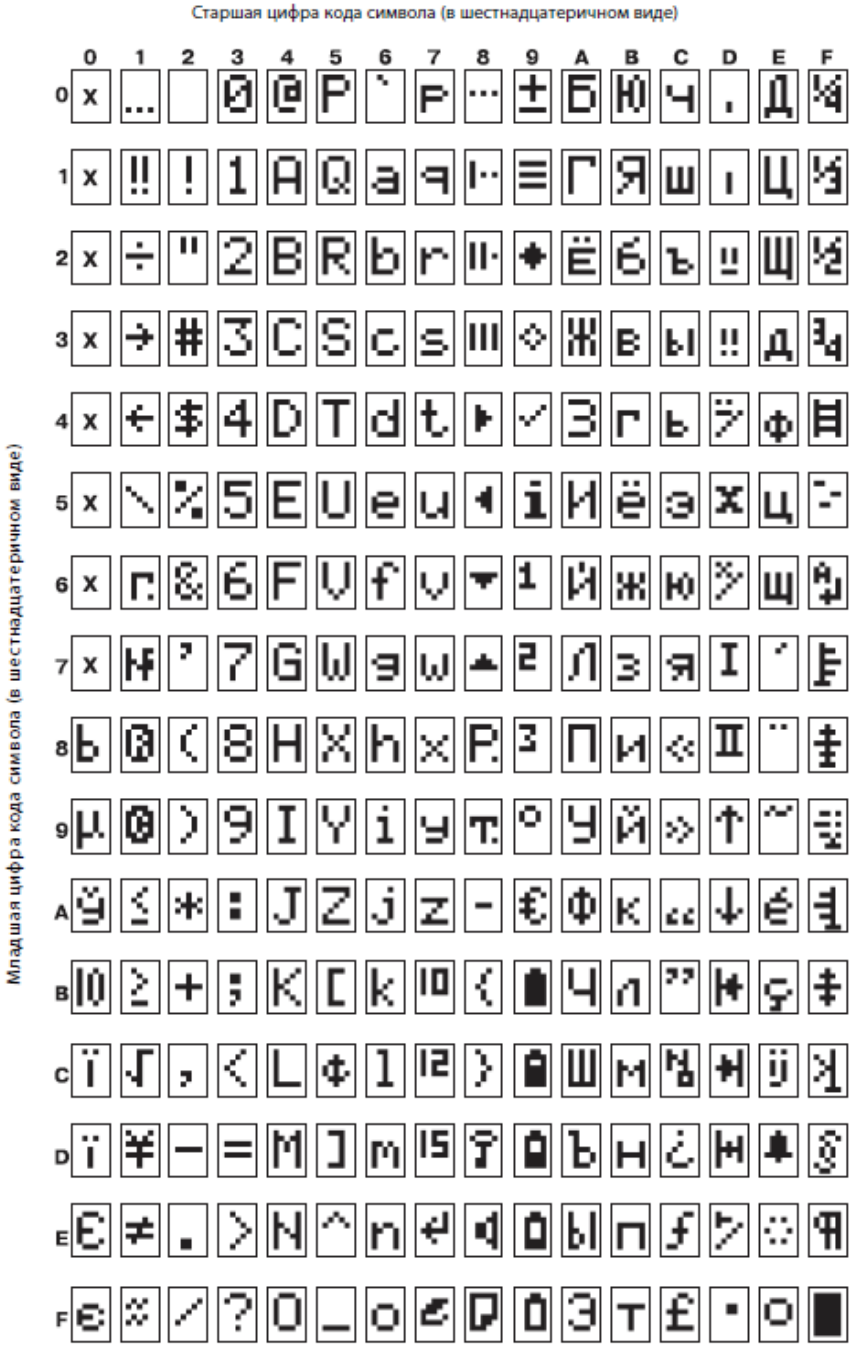
Библиотека предоставляет набор простых функций для работы с дисплеями. Рассмотрим некоторые из них.

- `Void lcd_init(uint8_t dispAttr)` — инициализирует дисплей. Эту функцию нужно вызвать один раз в начале программы.
  - `dispAttr` может иметь следующие значения:
    - **LCD_DISP_OFF** — дисплей выключен
    - **LCD_DISP_ON** — дисплей включён, курсора нет
    - **LCD_DISP_ON_CURSOR** — дисплей включён, курсор есть
    - **LCD_DISP_ON_CURSOR_BLINK** — дисплей включён, мигающий курсор
- `void lcd_clrscr (void)` — очищает дисплей
- `void lcd_home (void)` — возвращает курсор на исходную позицию
- `void lcd_gotoxy (uint8_t x, uint8_t y)` — устанавливает курсор в указанную позицию
- `void lcd_putc(char c)` — выводит символ на текущей позиции курсора. После вывода символа курсор переходит на следующую позицию
- `void lcd_puts(const char* s)` — выводит строку на текущей позиции курсора.
- `void lcd_command(uint8_t cmd)` — отправляет указанную команду контроллеру дисплея. Полный список поддерживаемых команд находится в документации на дисплей
- `void lcd_data(uint8_t data)` — посылает байт данных контроллеру дисплея

Этих несложных функций достаточно для того, чтобы реализовать вывод служебной информации, меню устройства и прочие интересные вещи. У дисплея также есть и другие возможности. Например, вы можете создавать свои собственные символы, которые неизвестны знакогенератору, встроенному в дисплей. Более подробно обо всём этом написано в документации.

Перекодировка кириллицы

В части, посвящённой вступлению в С, мы узнали про различные кодировки, которые используются для вывода символов, отличных от латинских. Существует несколько разных кодировок для вывода кириллицы, но, к сожалению, ни одна из не соответствует той, которая прошита в знакогенерторе LCD-дисплея. Ниже приведена таблица соответствия числовых значений и символов.



Как видите, символы с номерами от 0x20 до 0x7F соответствуют стандартной таблице ASCII. Это значит, что мы можем свободно использовать функции библиотеки дисплея в таком виде:

```
lcd_home();
lcd_puts("Hello world!");
```

В правой части таблицы находятся кириллические символы. Как видите, в ней не достаёт некоторых символов, таких как «А», «О» и т. д. — их можно взять из латинской части таблицы. Очевидно, что мы не можем просто написать

```
lcd_puts("Привет, мир!");
```

Так как кодировка, используемая компилятором, отличается от весьма оригинальной кодировки знакогенератора, на экране мы увидим совсем не то, что хотелось бы. Чтобы исправить ситуацию, есть два выхода. Первый — это задавать строковые константы в виде чисел (все помнят, что символ — это всего-навсего число?):

```
#include <avr/pgmspace.h>
#include "lcd.h"

PROGMEM const char S1[] = {0xA8, 'p', 0xB8, 0xB3, 'e', 0xBF, ',', ' ', 0xBC, 0xB8, 'p', '!'};

int main() {
       lcd_init(LCD_DISP_ON);
       lcd_puts(S1);
       return 0;
}
```

В этом примере мы выводим на экран строку «Привет, мир!» Заметьте, что кириллические символы, у которых есть эквиваленты в латинском алфавите, заменены ими и введены в виде символов. Чисто кириллические символы представлены шестнадцатеричными цифрами, подобранными в соответствие с приведённой таблицей.

Также обратите внимание на директиву препроцессора PROGMEM, которая находится перед объявлением массива. Эта директива становится доступной, если включить заголовок *<avr/pgmspace.h>* и означает, что данный массив будет помещён во флэш-память, в которой находится исполняемый код программы. Обычно переменные (в том числе, и массивы) располагаются в оперативной памяти, позволяющей изменять их значения. Однако в нашем случае строка является константой, и изменять её мы не будем. Поэтому было бы логично не занимать оперативную память, и поместить эту строку во флэш-память, которая не позволяет изменять своё содержимое во время исполнения программы (за исключением случая самопрограммирования, но он не в счёт).

Приведённый выше способ не очень удобен тем, что вам нужно долго и мучительно подбирать числовые значения, соответствующие кириллическим символам. Кроме того, такие строки становятся просто нечитаемы. Другим решением может оказаться перекодировка строк «на лету». Вы компилируете программу с нормальными кириллическими строками в привычной для вас кодировке, а некая функция, служащая обёрткой вокруг базовых функций библиотеки дисплея, будет перекодировать эту строку в вид, пригодный для знакогенератора. Однако помните, что этим вы ударите по производительности микроконтроллера, и, по сути, будете выполнять лишние действия (что-то, что можно сделать руками до компиляции, по-хорошему, надо делать руками до компиляции).

Пример

Рассмотрим небольшой пример использования библиотеки.

```
#include "lcd.h"

int main () {
      lcd_init(LCD_DISP_ON);
      while (1) {
        /* Очистить дисплей и вернуть курсор в начальную позицию */
        lcd_clrscr();
        /* Вывести на дисплей строку с переводом каретки ('\n') */
        lcd_puts("LCD Test Line 1\n");
        /* Курсор теперь на второй строке */
        lcd_puts("Line 2");
        /* Передвинуть курсор на символ 8 второй строки */
        lcd_gotoxy(7,1);
        /* Вывести один символ */
        lcd_putc(':');
        /* Вызвать некую функцию, которая ожидает нажатия кнопки */
        wait_until_key_pressed();
      }
}
```

В этом примере мы сначала инициализируем дисплей, а затем выводим несколько строк. После это вызываем некую функцию, которая ждёт нажатия кнопки. После нажатия кнопки всё происходит заново (так как код находится в бесконечном цикле).

Задания для лабораторной работы

Простые

1. Вывод заранее написанных строк в заранее заданных координатах с интервалом времени
2. Вывод бегущей строки с названием группы или своего имени/фамилии
3. Вывод бегущей строки с алфавитом
4. Реализовать простую анимацию. Паттерн — на своё усмотрение

Средние

1. Ввод цифр с клавиатуры. Две кнопки — для уменьшения/увеличения, две — для смены позиции
2. Ввод цифр в верхней строчке, одновременно отображать введенное в hex в нижней строке
3. Реализовать набор текста как смс в телефоне, но на 4 кнопках
4. Реализовать меню гипотетического устройства с вложенными пунктами. Пункты и навигация — на своё усмотрение

Сложные

1. Используя документацию на дисплей, разработать набор из несольких собственных символов с последующим выводом на дисплей.

2. Сделать перекодировку кириллицы «на лету». В программу строки зашиваются в нормальном виде; при выводе на дисплей — перекодируются.

Лабораторная работа №3

В последней лабораторной работе мы научимся работать с таймерами и прерываниями.

Прерывания

Практически в любом микроконтроллере существует система прерываний, позволяющая реагировать на определённые события. Таким событием может быть изменение логического уровня на выводе микроконтроллера, переполнение таймера, завершение работы АЦП и т. д. При наступлении такого события нормальный ход выполнения программы приостанавливается, и вызывается *обработчик прерывания*. При этом совершенно неважно, какая именно операция выполнялась в момент наступления прерывания: выполнение будет тут же приостановлено, и управление перейдёт к обработчику прерывания. После того, как обработчик завершит работу, выполнение программы возобновится с того места, на котором она была остановлена.

Включение прерываний

При запуске контроллера прерывания по умолчанию запрещены. Это значит, что ваша программа не сможет реагировать на события, даже если для них предусмотрены обработчики. Для включения прерываний используется функция *sei()*, которая наравне с остальными средствами работы с прерываниями находится в заголовочном файле *<avr/interrupt.h>*.

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

int main()
{
	sei(); // Разрешить прерывания
	return 0;
}
```

После вызова функции *sei()* ваши обработчики автоматически начнут обрабатывать прерывания, на которые они настроены.

Чтобы запретить использование всех прерываний, достаточно вызвать функцию *cli()*.

Важно запомнить, что при вызове обработчика прерывания все остальные прерывания автоматически отключаются, и включаются только после завершения данного обработчика. Это значит, что если во время обработки какого-то события произошло другое событие, оно не будет обработано. Такое поведение можно обойти и разрешить вложенные прерывания, однако, делать этого не рекомендуется из-за возможных проблем и трудноуловимых багов.

Внешние прерывания

В контроллерах ATMega16 предусмотрены три пина, которые могут генерировать *внешние прерывания.* Если разрешить эти прерывания, можно обрабатывать события, возникающие при изменении логического уровня на этих пинах.

Для включения внешних прерываний используются биты *INT0, INT1* и *INT2* регистра *GICR*. Если бит установлен в 1, будет включена обработка прерывания на соответствующем пине.

С помощью регистров *MCUCR* и *MCUCSR* можно настроить, при каких именно событиях будут вызываться прерывания.

За настройку прерывания *INT0* отвечают биты *ISC00* и *ISC01* регистра *MCUCR*. В таблице приведены различные сочетания их значений.

ISC01	***ISC00***	***Описание***
0	0	Прерывание вызывается при удержании на INT0 низкого уровня
0	1	Прерывание вызывается при любом изменении логического уровня на INT0
1	0	Прерывание вызывается по заднему фронту импульса на INT0
1	1	Прерывание вызывается по переднему фронту импульса на INT0

За настройку прерывания *INT1* отвечают биты *ISC10* и *ISC11* регистра *MCUCR*. В таблице приведены различные сочетания их значений.

ISC11	***ISC10***	***Описание***
0	0	Прерывание вызывается при удержании на INT1 низкого уровня
0	1	Прерывание вызывается при любом изменении логического уровня на INT1
1	0	Прерывание вызывается по заднему фронту импульса на INT1
1	1	Прерывание вызывается по переднему фронту импульса на INT1

Если для прерывания INT0 или INT1 установлен режим вызова по низкому уровню, соответствующий обработчик будет вызывать до тех пор, пока пин будет иметь низкий логический уровень.

Прерывание *INT2* настраивается с помощью бита *ISC2* в регистре *MCUCSR*. Это прерывание может реагировать только на смену логического уровня. Если бит *ICS2* установлен в «1», прерывание будет вызываться по переднему фронту импульса. Если *ISC2* имеет значение «0», то прерывание будет вызвано по заднему фронту.

Обработка прерываний

Для создания обработчиков прерываний используется макрос *ISR()*, определённый в заголовочном файле *<avr/interrupt.h>*. Внешний вид этого макроса похож на объявление функции:

```
ISR(INT0_vect)
{
	// Действия по обработке прерывания
}
```

В качестве параметра для этого макроса передаётся имя прерывания, которые вы хотите обработать. В данном случае обрабатывается внешнее прерывание INT0. Имена других прерываний можно найти в документации по libc, в разделе “Library Reference” → “Interrupts”.

Рассмотрим более полный пример. Предположим, что к порту A подключен модуль светодиодов, а к первой половине порта D — модуль кнопок. В этом случае третья кнопка будет соответствовать пину, альтернативной функцией которого является внешнее прерывание INT0.

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

ISR(INT0_vect) {
	// Проверяем, нажата ли кнопка, и, если да, включаем светодиод
	if (bit_is_set(PIND, PD2))
		PORTA |= _BV(PA0);
	else
		PORTA &= ~(_BV(PA0));
}

int main() {
	DDRA = 0xff; // Порт A на вывод
	PORTA = 0xff; // Высокий уровень на всех пинах порта A
	PORTD |= _BV(PD2); // Включаем подтягивающий резистор для кнопки

	MCUCR |= _BV(ISC00);	// Режим срабатывания прерывания INT0 – по
				// изменению логического уровня
	GICR |= _BV(INT0); // Разрешить прерывание INT0
	sei(); // Разрешить обработку прерываний

	while (1) {} // Бесконечный цикл

	return 0;
}
```

Как видите, в этом примере мы не проверяем состояние кнопки в главном цикле. Вместо этого, мы создаём обработчик прерывания, который будет вызываться только тогда, когда состояние кнопки действительно изменится. Это позволяет сэкономить ресурсы микроконтроллера и не тратить время на бесполезные проверки.

Таймеры

Таймеры позволяют отмерять интервалы времени и подсчитывать количество тактов. С помощью таймеров можно реализовать временную задержку, не занимая при этом ресурсы микроконтроллера (в отличие от функции *delay*). Также таймер может использоваться для генерации сигнала ШИМ.

Тактовая частота таймеров

В микроконтроллере ATMega16 есть три таймера: два восьмибитных (Timer0 и Timer2) и один шестнадцатибитный, Timer1. Каждый таймер работает независимо от

программы, выполняющейся в микроконтроллере, и не занимает системных ресурсов. По сути, таймер представляет собой обычный счётчик, который увеличивается на единицу при каждом тактовом импульсе. В качестве тактового сигнала можно выбрать такт микроконтроллера или же включить делитель частоты, который будет генерировать тактовый сигнал, в n раз меньший по частоте, чем тактовый сигнал микроконтроллера.

Рассмотрим работу таймера на примере Timer0.

Основной регистр, с помощью которого управляется Timer0 — это TCCR0. Три младших бита (CS00, CS01 и CS02) определяют тактовую частоту, на которой работает таймер. Как только значение хотя бы одного из этих битов изменяется, таймер начинает работать.

CS02	***CS01***	***CS00***	***Описание***
0	0	0	Таймер выключен
0	0	1	Частота контроллера (в данном курсе — 4 МГц)
0	1	0	Частота контроллера / 8
0	1	1	Частота контроллера / 64
1	0	0	Частота контроллера / 256
1	0	1	Частота контроллера / 1024
1	1	0	Внешний тактовый генератор на пине T0. Так по заднему фронту.
1	1	1	Внешний тактовый генератор на пине T0. Так по переднему фронту.

После включения таймера каждый тактовый импульс, поступивший к нему, вызовет увеличение значения регистра **TCNT0** на единицу. Регистр TCNT0 имеет размер в 1 байт (отсюда и разрядность таймера). При достижении максимального значения (для восьмибитного таймера — 255), значение регистра будет автоматически сброшено в ноль. При этом можно разрешить специальное прерывание, которое будет вызываться при каждом переполнении таймера.

Для управления прерываниями всех таймеров используется регистр TIMSK. Биты TOIE0 и OCIE0 отвечают за прерывания переполнения и сравнения для Timer0. Если бит установлен в 1, соответствующее прерывание будет вызываться.

Режимы работы таймеров

Все три таймера, имеющихся в контроллере ATMega16, могут работать в четырёх основных режимах. Режим работы выбирается с помощью битов *WGMx0* и *WGM0x1* регистра *TCCRx*. Рассмотрим таблицу режимов на примере Timer0.

Номер	***WGM01***	***WGM00***	***Режим работы***	***Макс. значение TCNT0***
0	0	0	Нормальный	0xFF
1	0	1	ШИМ, синхронный по фазе	0xFF
2	1	0	CTC	*OCR0*
3	1	1	Быстрый ШИМ	0xFF

В нормальном режиме работы таймер увеличивает значение регистра TCNT0 на единицу на каждом тактовом импульсе. Значение увеличивается до 255 (предел 8-битного регистра), после чего сбрасывается в 0. В режиме *CTC* таймер считает не до 255, а до значения, указанного в регистре *OCR0*. После достижения *TCNT0* этого значения он сбрасывается в 0. В режимах ШИМ таймер генерирует специальный сигнал, позволяющий, в числе прочего, управлять напряжением. Более подробно все эти режимы мы рассмотрим далее.

Организация временной задержки (нормальный режим)

Рассмотрим пример, в котором с помощью таймера реализуется временная задержка в одну минуту. Для начала нам придётся рассчитать тактовую частоту, с которой должен работать наш таймер.

Очевидно, что какой бы делитель частоты мы не выбрали, мы не сможем уложить количество тактов, требуемое для 60-секундной задержки, в 8-битный регистр TCNT0. Значит, нам придётся заводить отдельную переменную-счётчик и увеличивать её на единицу каждый раз, когда таймер отсчитает 256 тактов и переполнится. Будем использовать для счётчика переменную типа *int*.

Предположим, что в качестве тактового сигнала мы возьмём такт микроконтроллера, то есть, не будем использовать делитель. Тогда количество времени, которое пройдёт за один такт, будет равно:

Теперь разделим необходимое нам время задержки (60 с.) на это значение, и получим количество тактов, которое будет равно одной минуте:

$$\frac{60}{0.00000025} = 240000000$$

Так как мы будем считать не каждый первый такт таймера, а только каждый 256-й, разделим это значение на 256:

$$\frac{240000000}{256} = 937500$$

Здесь мы получили окончательное значение, при достижении которого нашей переменной-счётчиком мы можем сказать, что прошла одна минута. Однако полученное значение гораздо больше верхнего предела типа *int*, который составляет 65535. Решить эту проблему можно двумя способами: сделать переменную-счётчик типа *long* или включить делитель частоты для таймера, тем самым увеличив интервал времени, соответствующий одному такту таймера. Первый вариант менее предпочтителен, так как занимает больше памяти, тогда как второй вариант не требует никаких дополнительных ресурсов. Если мы будем делить частоту контроллера на 256, то нам понадобится

$$\frac{937500}{256} = 3662{,}109$$

переполнений таймера. Если немного пожертвовать точностью и округлить это число до 3662, оно прекрасно впишется в пределы значений типа *int*.

Итак, можно записать обобщённую формулу для вычисления значения делителя частоты и максимального значения переменной-счётчика:

$$\frac{t * f}{n * 256} = k$$

Где t — необходимое время задержки, f —частота микроконтроллера, n —делитель частоты, k —значение переменной-счётчика, достижение которого означает истечение заданного интервала. Чтобы воспользоваться этой формулой, для начала подставьте вместо n единицу. Если k получилось больше, чем максимально допустимое значение для типа переменной-счётчика, возьмите другое n из таблицы предыдущего параграфа. Изменяйте n до тех пор, пока не получите k, дающее наименьшую погрешность при округлении.

Теперь рассмотрим код, реализующий задержку в одну минуту.

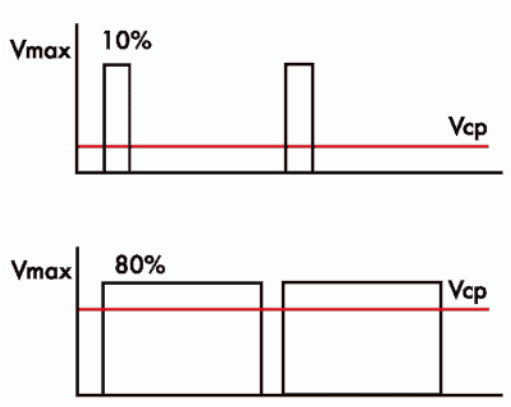
```
#include <avr/io.h>
#include <avr/interrupt.h>

#define MAX_CNT           3662   // Количество переполнений таймера для
                                 // интервала 60 с.
int cnt = 0;                     // Переменная-счётчик

void OneMinute();

void OneMinute() {
      TCCR0 &= ~(_BV(CS02));     // Выключим таймер, очистив бит CS02
      DDRA = 0xff;               // Порт A на вывод
      PORTA = 0x7f;              // Выключить все диоды, кроме последнего
}

ISR(TIMER0_OVF_vect) {    // Обработчик прерывания переполнения таймера
      cnt++;              // Увеличим переменную-счётчик
      if (cnt == MAX_CNT) // Если достигли нужного кол-ва прерываний,
            OneMinute(); //  вызовем некую функцию
}

int main() {
      sei();                // Разрешить обработку прерываний
      TIMSK |= _BV(TOIE0); // Разрешить прерывание по переполнению Timer0
      TCCR0 |= _BV(CS02);  // Запустить таймер с делителем частоты 256

      while (1){} // В цикле можно делать полезную работу

      return 0;
}
```

Как видите, здесь мы сначала разрешаем прерывание по переполнению Timer0, а затем запускаем его с делителем 256. Когда значение таймера сбрасывается с 255 в 0, вызывается обработчик, увеличивающий значение переменной-счётчика и проверяющий её значение. Если мы достигли количества прерываний, соответствующего одной минуте, вызывается функция, которая останавливает таймер и зажигает диод на модуле, подключённом к порту A.

Широтно-импульсная модуляция

Как известно, в цифровой электронике применяются дискретные уровни сигнала. Это значит, что в любой момент времени допустимые значения описываются только тремя основными состояниями: высокий логический уровень, низкий логический уровень и неопределённое состояние. Мы не можем выдать на выводе микроконтроллера напряжение, например, 3.5 В с помощью стандартных средств. Для решения этой проблемы можно использовать метод широтно-импульсной модуляции.

Метод заключается в формировании периодического сигнала с постоянным периодом и переменной скважностью (отношением периода к длительности импульса).



Из рисунка видно, что у этого сигнала период остаётся неизменным, тогда как длительность импульса со временем меняется. Если такой сигнал подать на вход фильтра нижних частот с правильно подобранными параметрами, то на выходе этого фильтра мы получим напряжение, усреднённое по периоду сигнала.

Рассмотрим пример. Допустим, у нас есть сигнал с амплитудой 5 В и неким периодом T. Пусть длительность импульса составляет 0.1 * T. В таком случае, половину времени, занимаемого одним периодом импульса, сигнал будет иметь значение 5 В, а другую половину — 0 В. Значит, среднее значение напряжения будет равно 0.5 В. Если же мы изменим длительность импульса на 0.8 * T, среднее напряжение будет приблизительно равно 4 В.

В этом и заключается принцип широтно-импульсной модуляции: изменяя скважность цифрового сигнала, мы можем получить любое напряжение, отличное от напряжения цифровых уровней.

В микроконтроллере ATMega16 таймеры имеют встроенную поддержку ШИМ. Работает она следующим образом: при достижении таймером значения 0 логический уровень на специальном выводе устанавливается в «1». При достижении TCNTx значения, указанного в регистре OCRx, вывод переключается на низкий уровень. OCRx — это специальный регистр, в который можно записать значение от 0 до 255 (или от 0 до 65535 в 16-битном таймере). Как только значение регистра TCNTx, который считает импульсы, генерируемые таймером, достигает значения, указанного в OCRx, происходит некоторое событие, которое зависит от режима работы таймера. Как уже было сказано, в режиме ШИМ таким событием является переключение пина OCx в значение «0». Таким образом, изменяя значение регистра OCRx, можно изменять скважность сигнала, который автоматически генерируется микроконтроллером на выводе OCx.

Рассмотрим небольшой пример, который заставляет светодиод плавно включаться и выключаться. В этом примере мы подаём сигнал ШИМ прямо на светодиод, не используя фильтр. Мы можем так сделать из-за того, что частота переключения светодиода будет очень высока и, вследствие инертности нашего зрения, нам просто будет казаться, что он горит ярче или тусклее. В реальной жизни даже сигнал, получаемый после фильтра нижних частот, не будет идеально ровным, поэтому кроме ФНЧ рекомендуется использовать ещё и катушку индуктивности для сглаживания получаемого напряжения.

В этом примере мы будем использовать Timer0 и подключим модуль светодиодов к порту B. Дело в том, что выводы ШИМ таймеров чётко закреплены за определёнными

пинами в качестве альтернативных функций, и мы не можем вывести сигнал ШИМ на тот пин, который покажется нам более удобным (конечно, мы всегда можем генерировать сигнал ШИМ «руками», без помощи таймера). В случае использования Timer0, вывод OC0 закреплён за пином PB3.

Для включения режима ШИМ нужно установить в «1» биты *WGM00* и *WGM01* в регистре *TCCR0*. Кроме того, нам нужно включить пин OC0, установив в «1» бит *COM01* в этом же регистре.

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

unsigned char delta = 1;

ISR(TIMER0_OVF_vect) {
	OCR0 += delta;
	if (OCR0 == 255)
		delta = -1;
	else if (OCR0 == 0)
		delta = 1;

}

int main() {
	sei();
	DDRB |= _BV(PB3);
	TIMSK |= _BV(TOIE0);
	TCCR0 |= _BV(WGM00) | _BV(WGM01) | _BV(COM01) | _BV(CS01) | _BV(CS00);

	while (1) {}
	return 0;
}
```

Этот код очень похож на код из предыдущего примера. Отличие заключается в том, что при записи значения в регистр TCCR0 мы включили режим ШИМ и установили делитель частоты на 64. Кроме того, в обработчике прерывания переполнения мы изменяем значение регистра OCR0. Как видно из кода, значение OCR0 сначала линейно увеличивается до 255, а потом постепенно уменьшается до 0. Этим мы изменяем скважность сигнала ШИМ, генерируемого таймером и заставляем светодиод плавно загораться и гаснуть.

Режим CTC

Этот режим практически ничем не отличается от нормального, за исключением того, что таймер увеличивает значение счётчика TCNTx не до максимального значения, а до значения, записанного в OCRx. Эту возможность можно использовать для организации более точной задержки в тех случаях, когда число *k* из формулы расчёта частоты таймера получается дробным, а округление по каким-то причинам недопустимо.

Выводы OC(n)x

Выводы микроконтроллера, которые мы использовали в режиме ШИМ, могут использоваться и в других режимах работы таймера. При совпадении значения счётчика TCNTx со значением, записанным в регистре OCRx, происходит прерывание, которое вы можете разрешить, установив в «1» бит OCIE(n)x в регистре TIMSK. Кроме того, это

совпадение может изменить состояние вывода OC(n)x в соответствие со следующей таблицей (для Timer0 и регистра TCCR0):

COM01	***COM00***	***Описание***
0	0	Нормальный режим работы порта, OC0 не используется
0	1	OC0 изменяет состояние на противоположное
1	0	OC0 устанавливается в «0»
1	1	OC0 устанавливается в «1»

Задания

Простые

1. Реализовать любое из лёгких заданий лабораторной работы №1 при помощи таймеров
2. Реализовать частотомер на прерывании и таймере
3. Генератор меандра заданной частоты
4. Реализовать часы на дисплее

Средние

1. Управление яркостью светодиода с помощью кнопок
2. Реализовать делитель частоты с заданным коэффициентом деления
3. Реализовать делитель с изменяемым коэффициентом деления, значение выводить на дисплей

Сложные

1. На вход внешнего прерывания подаётся хитрый сигнал. Определить длительность импульса, период и скважность; результат вывести на дисплей.
2. Реализовать передачу массива чисел по ИК. На одном стенде собирается приёмник, на другом — передатчик. На приёмном стенде принятая последовательность выводится на дисплей.