# Программирование на WinAVR Си
# зарядного устройства (ZU v1.7) на AT Mega32.

## Вступление.

Несмотря на то, что тема раскрыта не полностью (слишком большой замах), в статье вы найдете ответы на многие вопросы, а на остальные вопросы ответят сами исходники программы.

Данная статья адресована тем энтузиастам, которые решились собрать ZU v1.7 (универсальное зарядное устройство), описанное на http://avrcpp.narod.ru Главной задачей этой статьи является простыми словами объяснить разработку программ для микропроцессоров на WinAVR Си и прийти к выводу, что многое, что кажется сложным и запутанным, всего лишь - стремление максимально просто достичь желаемого результата.

Сложные куски программ, приведенные ниже, и особенности языка программирования, не являются главными моментами статьи. На них можно не обращать внимание. Просто внимательно рассматривайте вставки кода и читайте комментарии к нему, не пытаясь понять и запомнить сам код.

Набив многочисленные шишки и истратив много времени на поиски, я хочу рассказать, как я себе представляю программирование. Я надеюсь, что стремление ничего не упустить и остаться понятным, не слишком раздуло статью. Некоторая доля философских рассуждений немного отвлечет и расслабит читателя для лучшего запоминания.

## Урок №1 (Устанавливаем WinAVR)

WinAvr – одна из самых лучших программ для программирования процессоров Atmel. И вот мои доказательства на момент написания статьи:

1. Это бесплатный компилятор Си, Си++, распространяемый по лицензии GNU. И это наиглавнейший плюс, для программистов, желающих не нарушать лицензионное законодательство. «Embedded IAR C++ for AVR» стоит запредельно высоко для самодельщиков >$1000. Про остальные компиляторы точно не скажу $150 - $300, но проблема с ними не в цене, а в возможности их приобрести.
2. Скорость скомпилированной программы находится почти на уровне лучших компиляторов.
3. Размер скомпилированной программы вполне приемлемый.
4. Явных, непреодолимых ошибок не обнаружено.
5. В программе имеется своя оболочка – текстовый редактор «Programmers Notepad 2» из которого можно компилировать и прошивать процессоры, например для программатора PonyProg.
6. Данная программа написана на WinAVR-20071221rc1. В более поздней версии оптимизация по размеру сделана хуже.
7. Этот компилятор продолжает совершенствоваться.

Но есть также и минусы: оптимизации по размеру хотелось бы больше, тем более, что на других компиляторах тот же текст дает меньший размер программы. Работа с отдельными битами упразднена с мотивировкой увеличения скорости, не совсем логичное использование глобальных переменных, затруднена работа с указателями на константы во FLASH памяти, библиотека функций не блещет разнообразием. И, все же, бесплатность и доступность перевешивает все эти недостатки, которые не лишают нас возможности довести проект до конца и реализовать все наши задумки на должном уровне.

Как установить:

1. Закачиваем установочный файл с сайта WinAvr.

2. После установки запускаем «c:\winavr\pn\pn.exe» это редактор с возможностью компилирования и прошивки.

3. Редактор необходимо настроить на компилятор и утилиту прошивки AVRDUDE, добавив в него несколько команд:

- **ОЧИСТКА:** Tools => Options => Ветка Tools => Выбрать в списке: C/C++ => ADD => Name=**CLEAR**; Command= C:\WinAVR\utils\bin\make.exe; Folder= %d; Parameters=clean
- **КОМПИЛЯЦИЯ:** Tools => Options => Ветка Tools => Выбрать в списке: C/C++ => ADD => Name=**COMPIL**; Command= C:\WinAVR\utils\bin\make.exe; Folder=%d; Parameters=all
- **ПРОГРАММИРОВАНИЕ:** Tools => Options => Ветка Tools => Выбрать в списке: C/C++ => ADD => Name=**BURN**; Command= C:\WinAVR\utils\bin\make.exe; Folder=%d; Parameters=program

4. В папке с исходниками ЗУ есть «Makefile». В любом проекте WinAvr должен быть такой файл. Обратите внимание на следующие параметры:

- F_CPU = 16000000 - тактовая частота
- FORMAT = ihex - формат файла прошивки
- TARGET = main - название файла прошивки
- OPT = s - оптимизация по всему
- AVRDUDE_PROGRAMMER = ponyser - подключен программатор PonyProg через COM-порт. Еще раз напоминаю используйте только честный COM с материнских плат и не используйте переходники USB to COM
- AVRDUDE_PORT = com1 - номер COM-порта
- AVRDUDE_WRITE_FLASH = -U flash:w:$(TARGET).hex
- AVRDUDE_WRITE_EEPROM = -U eeprom:w:$(TARGET).eep

5. Остальные настройки «Makefile» без особой надобности не трогайте!

6. Для запуска редактирования проекта ЗУ необходимо запустить «c:\winavr\pn\pn.exe zu.pnproj».

7. Для того чтобы скомпилировать проект необходимо сначала выполнить команду Tools => CLEAR затем Tools => COMPIL

8. Чтобы прошить ЗУ новой скомпилированной прошивкой через программатор PonyProg, разведенный на цифровой схеме ЗУ, надо выполнить команду: Tools => BURN. К сожалению, мне не удалось прошить фузы (биты конфигурации процессора) из WinAvr, а без них работать не будет, поэтому сначала надо на программе от PonyProg прошить фузы, а потом уже шить программу на WinAvr.

Теперь у нас есть язык программирования. Теперь мы можем писать программы для процессоров AT MEGA.

## Урок №2 (общий взгляд):

Мы сделали зарядное устройство с процессором, но без прошивки оно работать не будет. Процессор должен знать как и в какой последовательности нужно заряжать аккумуляторы различных типов. Для написания микропрограммы необходимо в общих чертах представлять себе как работает процессор. Это позволит нам выбрать язык программирования и аккуратно его использовать. Нужно сформулировать принцип действия нашего ЗУ: сколько времени тратится на различные операции. Что главное, что второстепенное. Нужно знать порядок значений, скоростей, времени выполнения различных операций.

Общую картину все время держим в голове и каждое принятое решение прикладываем к картине на предмет совместимости. Это позволит нам избежать противоречивых решений и излишней работы, ведущей в тупик. Но конечно же бывает

всякое. Учесть все и не наделать ошибок — это сложная задача.

ATMEGA32 - это не просто микросхема, это целый компьютер в одном флаконе. Внутри него есть основные составляющие любого компьютера:

а) Процессор, АЛУ (исполнитель программы)

б) Память (хранилище программы) ОЗУ=Оперативное Запоминающее Устройство, ПЗУ=Постоянное Запоминающее Устройство

в) Порты ввода-вывода (воздействие на внешний мир и получение информации из внешнего мира)

Этих составляющих достаточно, чтобы описать любой из компьютеров мира. Все многообразие компьютерных программ от тетриса до глобальной сети Интернет работает на одинаково устроенных компьютерах. Все программы всего мира делают одно и тоже:

а) считывают данные из памяти (или портов ввода/вывода) в процессор

б) выполняют арифметические действия

в) записывают результат в память (или порты ввода/вывода)

Мы люди имеем представление обо всем в виде образов, многие образы обозначены словами, слова состоят из букв. Буквы перенумерованы. Например 128 это "А", 129 это "Б" и т.д. Если мы посчитаем сколько знаков, включая буквы большие и маленькие, знаки препинания, цифры и английские буквы и всякие простенькие знакосимволы, словом все, что может понадобиться, то выйдет около 250 штук. Выходит весь мир и все объекты в нем можно закодировать числами.

В десятичной системе десять цифр, а в двоичной две - 0 и 1. И двоичная и десятичная система отражают одну действительность, они представляют количество или номер по порядку. Каждая система представляет по своему, но имеет ввиду один и тот же смысл. Двоичная система более близка схемотехнике, потому что 0 и 1 можно представить как 0 вольт и +5 вольт на ножках микросхем.

Во всех компьютерах мира на ножках процессоров, микросхем памяти, различных драйверов, северном и южном мосте вашего компьютера, внутри всех микросхем в ячейках памяти, внутри процессора в регистрах и логических схемах происходит одно и тоже событие — меняется напряжение: то ноль, то +5в (или какое либо другое постоянное напряжение). Так живет программа внутри компьютера, внутри мобильного телефона, внутри принтера, факса, ксерокса, в DVD проигрывателе, в интернет сетях, в спутниковом оборудовании, во всех цифровых системах. Именно так и работает компьютер. Именно так компьютер создает картину мира - в виде цифр 0 и 1, собранных в числа, которыми закодированы все объекты мира.

Реальность это изменение напряжений в ячейках памяти. То что мы видим на экране компьютера — это иллюзия.

**То что мы видим вокруг себя и чувствуем — это иллюзия, реальность — это взаимодействие частиц, которые даже не являются твердыми шариками, а являются вероятностной энергетической волной или искажением структуры вакуума, пространства и времени.**

## Урок №3 (Языки программирования)

Все языки программирования суть одно и тоже - передать исполнителю (процессору) кратко, однозначно, понятно то, что необходимо делать.

Теперь рассмотрим это утверждение с разных сторон. Программы существуют с доисторических времен. Программы зашиты на генном уровне в каждой клетке, у всех животных и у человека. Инстинкты - это и есть программы (BIOS человека). Мы на 99.9 процентов запрограммированы и действуем по программе. Практически все, что делает человек делается по аналогии, как научили родители, школа, институт, двор, бандитская шайка, кружок творчества, реклама, инструкция по применению. Почти все мои знания

внедрены в меня обществом. Хотя у каждого есть иллюзия, что знания свои и что они абсолютно верны. У меня есть причины подозревать, что большая половина моих знаний и мнений ошибочна. Но эти знания общеприняты, что несколько успокаивает.

Однажды я задумался: А что собственно сделал (придумал) лично я? И не смог сходу ответить на этот вопрос. Идеи я взял в интернете, способы спросил у знающих людей и в книжках прочитал. Что такого сделал я? Перекомпоновал чужие идеи-программы, состыковал в другой последовательности, сделал несколько выборов что лучше что хуже - вот и все мое творчество, остальное только работа и напряжение по перебору и подгонке.

Выходит удел любого человека **делать правильный выбор и прикладывать усилия**. Результат - это подарок мира. Результат однозначно не зависит от меня. Я захотел играть на скрипке, в доску расшибся, но не смог, нет у меня слуха. Выходит результат не может быть оценкой меня и моих способностей. Наивысшая оценка меня как личности - это оценка правильности моего выбора и оценка вложенных усилий. Не смотря на то, что многие люди достигают задуманного результата, нельзя считать, что это их достижение. Доказательство очень простое: представим что Солнца нету и вот никто и не может получить вообще никакого результата.

Каждый процессор «понимает» только свой язык (язык машинных кодов), поэтому программа будет написана только на этом языке. Процессоры INTEL8086 (IBM XT), Z80 (ZX SPECTRUM), MOTOROLA 6502 (APPLE или АГАТ) каждый понимают только свои машинные коды, потому что устройство процессоров разное.

Для того чтобы человеку было удобнее понимать язык машинных кодов, каждому коду поставлена в соответствие команда/слово или аббревиатура, содержащие краткий смысл команды. Такое улучшение понимания машинных кодов называется ассемблер. Никто не пишет программы на машинных кодах, это очень не удобно, но есть ограниченное количество людей, которые пишут на ассемблере.

Ассемблер и машинные коды это почти одно и тоже. Одна команда ассемблера эквивалентна 1-3 машинным кодам. Язык ассемблера это самый простой и самый сложный язык на свете, это тот язык который не исчезнет, пока существуют процессоры. На ассемблере писать очень сложно в смысле воплощения общей идеи программы в элементарные команды, и одновременно очень легко в смысле понимания самих команд (элементарнее этих команд ничего не бывает).

Видов команд ассемблера как правило бывает очень мало: команды чтения и записи в память, команды чтения и записи в порты ввода вывода, команды стека (работа со сдвиговой памятью), команды безусловных и условных переходов, арифметические команды.

Вот как выглядит ассемблер (не пытайтесь понять и не ищите смысла это просто пример):

```
Машинный код    Ассемблер          Комментарий

38 100          MOVE X, 100    // Записать 100 в регистр X
57 3            ADD X, 3       // Добавить 3 к тому что лежит в регистре X
94              IN A           // Считать данные из порта ввода/вывода A
92 20           OUT B,20       // Записать в порт ввода/вывода B 20
85 00 10        JMP 1000       // Выполнять программу с адреса 1000
47 200          JZ 200         // Если значение было 0 перейти на 200
```

Писать программу на ассемблере это все равно что объяснять человеку какие мышцы (а их от 200 до 500 в зависимости от способа подсчета) надо напрячь а какие

расслабить для совершения ходьбы.

Язык Си более высокого уровня. На нем гораздо легче и понятнее писать программы. Язык Си может все, что может ассемблер. Язык Си почти такой же быстрый как и ассемблер. За 20 лет программирования на Си я так и не исчерпал всех его возможностей и многого по прежнему не знаю. Я считаю, что оптимальное программирование это знание ассемблера в общих чертах и работа на Си.

Все остальные языки очень разные (например: fortran, pl1, basic, algol, cobol, refal, pascal, php, lisp, html, perl, prolog, java, sql). Некоторые языки удобны для интернета, некоторые для математики, некоторые для баз данных, некоторые языки служат для написания компиляторов. Но все эти языки в конечном счете создают программу на машинных кодах, которую выполняет процессор. Компиляторы (преобразователи в машинный код) многих из вышеуказанных языков написаны на Си. Язык Си может все, он универсален.

## Урок №4 (Структура программы на Си)

Когда-то давно, когда компьютеры еще не были столь производительны, компиляторы Си работали очень медленно. С целью легкости обнаружения ошибок, на текст программы Си было наложены ограничения, некоторые из них сохранились до сих пор.

Все программы на Си должны иметь приблизительно такой вид:

```
Команды препроцессора - команды для автоматического редактирования
текста программы перед компиляцией
Описание переменных, видных во всех подпрограммах

Подпрограмма1
  {
  Описание переменных, видных только внутри подпрограммы
  Тело подпрограммы
  }
...
ПодпрограммаN
  {
  Описание переменных, видных только внутри подпрограммы
  Тело подпрограммы
  }
Главная программа (запускается при включении или перезапуске ЗУ)
  {
  Описание переменных, видных только внутри подпрограммы
  Тело подпрограммы
  }
```

Придерживаться такого вида программы - это стильно, это удобно, это стандартно, это понятно. Придерживаться стиля это признак хорошего тона. Другие программисты скажут вам спасибо, так как при сосредоточении на смысле программы внешний вид несет дополнительную информацию и, тем самым, помогает программировать безошибочно.

Откроем исходники программы и попробуем разобраться что к чему. Есть ли там этот стиль? Где главная программа? Где подпрограммы? Какие правила для написания подпрограмм?

Вот она программа для записи в микропроцессор, написанная на языке Си (содержимое файла main.cpp из исходников Здесь):

```
#include <avr/io.h>  // Подключаем файл с подпрограммами и переменными от avr
#include "util.h"    // Подключаем файл со всеми описаниями переменных
#include "util.cpp"  // Подключаем файл со всеми подпрограммами
int main(void)       // Главная программа
{
#include "init.cpp"          // Инициализация всех ног процессора
#include "zagruzka.cpp"      // Загрузка начальных переменных из ПЗУ
while(true)                  // Бесконечно повторяем {содержимое скобок}
  {
  wdt_reset();               // Сброс собаки (авторесета) на случай зависания
  #include "sh.cpp"          // Прорисовка шаблонов - всех изображений ЖКИ
  #include "test.cpp"        // Рассчитываем все переменные по каналам
  if(iK1!=iK0)               // Если есть необработанные кнопки рисуем меню
    {
    #include "menu.cpp"      // Обработка кнопок
    }
  TestMainParam();           // Проверяем не горит ли чего
  if(Ch1.C)Go(Ch1);          // Если запущен канал 1 обрабатываем этот канал
  if(Ch2.C)Go(Ch2);          // Если запущен канал 2 обрабатываем этот канал
  #include "uart_in.cpp"     // Если надо чтонибудь получить с COM-порта
  #include "uart_out.cpp"    // Если надо чтонибудь послать на COM-порт
  }
}
```

Теперь я постараюсь объяснить каждую букву в этом тексте.

**#include <avr/io.h>** - подключает к моему файлу (вставляет весь файл **io.h** вместо этой строки) необходимые мне описания и функции из комплекта поставки WinAVR. Это очень удобно. Мне не надо писать свои стандартные подпрограммы, т. к. я могу воспользоваться оными из файла **io.h**. Угловые скобки говорят о том что файл надо искать в том месте куда установился WinAVR в подкаталоге **avr.**

**#include** и все что начинается на "#" это команды **пре**процессора, т.е. команды которые выполняются **перед** компиляцией программы. Перед компиляцией в текст программы добавляется содержимое всех включаемых файлов. Если бы я сам добавил все эти файлы, то программа разрослась бы и было бы трудно все это листать и понимать, а так это, всего лишь, одна строка. Эта команда является реализацией принципа - "Разбей сложное на много простых (логически обособленных) частей".

**#include "util.h"** и **#include "util.cpp"** - это мои файлы с описателями и функциями (полезные подпрограммы — утилиты, один раз написаны и используются мной для разных проектов). Двойные кавычки указывают компилятору, что файлы надо искать в той же папке где находится главный файл программы main.cpp. Расширение ".cpp" говорит о том, что внутри файла лежат подпрограммы на языке Си. Расширение ".h" говорит, что там лежат описания переменных и функций. В принципе, компилятору все равно что лежит внутри файлов и он это не отслеживает, но это нужно мне.

Есть еще одна широко используемая команда препроцессора: #define aaa bbb. Эта команда перед компиляцией обыскивает весь файл программы ниже этой команды и меняет aaa на bbb. Это тоже очень удобная команда, которую я использую для улучшения читаемости текста. Вместо сложного выражения в тексте программы просто ставлю какое-нибудь понятное слово, а перед компиляцией оно заменяется на сложное выражение.

Все команды препроцессора включая #include и #define напрямую никак не отражаются в конечной программе на машинных кодах, они просто меняют текст программы перед компиляцией. Сами слова #include и #define это жестко зафиксированные слова в Си. Их никак иначе нельзя использовать. Есть и много других слов которые жестко зафиксированы.

После компиляции Си программы получается файл main.hex это файл с машинными кодами. Как там все устроено нас не волнует. Важно только то, что у нас есть

главная программа main(), которая запустится сразу после включения ЗУ или после нажатия на кнопку RESET, если у вас такая есть.

До сего момента я все время путался с названиями: программа, подпрограмма, функция. По большому счету это одно и тоже. В Си принято использовать слово функция.

Попробуем сформулировать определение как можно проще. Функция это подпрограмма. Внутрь функции можно передавать параметры. После окончания функция может возвращать одно вычисленное значение. Приведем несколько примеров:

Пример 1: **wdt_reset();** - это вызов функции сброса сторожевой собаки. Внутри процессора ATMega32 есть специальный счетчик - «собака». В зависимости от настроек приблизительно раз в 2 секунды он перезапускает процессор. Если этот счетчик не сбрасывать, то процессор будет каждые 2 секунды перезапускаться. Этот механизм предусмотрен разработчиками ATMEL для предотвращения зависания программы. Всякое бывает, может компилятор глючит, может сама программа с ошибками, может железо процессора не работает от перегрева или радиации, но программы иногда зависают. В этот момент могут происходить непредсказуемые события, ЗУ может само себя спалить. Если программа зависла, то она не проходит основной цикл и не сбрасывает собаку, и тогда процессор перезапустится. Все процессы в ЗУ инициализируются как будто вы только что включили ЗУ. Конечно, надо искать ошибку и не допускать произвольных перезагрузок. И все же сторожевая собака должна быть.

Возвращаемся к функциям. Данная конкретная функция **wdt_reset();** не получает ни каких параметров (поэтому внутри скобок ничего нет) и ничего не возвращает, поэтому результат работы функции не присваивается никакой переменной.

Чтобы иметь право вызвать на выполнение эту функцию, надо чтобы ее текст был где-то выше вызова описан. Эта функция была написана внутри библиотек функций компилятора WinAVR, т.е. досталась нам готовенькой, и, чтобы мы могли ее вызвать, мы должны сказать компилятору, что это за функция. Для этого надо в начале нашего файла подключить файл описателей включающий в себя описание функций собаки **#include <avr/wdt.h>**. В этом файле описана эта функция и ее местонахождение в исходниках или представлен полный ее текст.

Пример 2: **x=sin(5); sin** - это функция, которая получает параметр 5 и возвращает результат в переменную x.

Пример 3: **int main(void)**{} **main** - это функция которая не имеет параметров. Об этом говорит слово void или пусто внутри круглых скобок. **main** возвращает как результат целое число об этом говорит слово **int** (специальное слово для описания типов переменных). Функция **main** вызывается один раз прямо там где написано ее тело, поэтому в программе больше нигде не встречается слово **main**.

Каждая функция, использованная в нашей программе, либо написана нами лично и помещена где-то в тексте (выше или ниже вызова) или заимствована из библиотеки стандартных функций WinAVR. Если тело функции ниже вызова, то выше вызова надо ее кратко описать (сообщить ее название, параметры и возвращаемое значение). Если вы в тексте программы увидели слово из английских букв и цифр (первая обязательно буква или "_") и сразу после него круглые скобки, знайте это функция.

Процессор - исполнитель программ на машинных кодах, работает как рабочий, действующий по плану работ. Он выполняет все по порядку слева направо и сверху вниз. Читает одну команду и выполняет, читает другую и выполняет. Это относится к нашему процессору ATMega32. Этот процессор не имеет никаких параллельных потоков выполнения. Существует одна точка текущего выполнения - один поток выполнения.

Только некоторые счетчики, COM-порт и АЦП работают сами по себе - аппаратно. Вспоминаем, что ATMega32 это маленький компьютер в одной микросхеме (в одном чипе). Внутри него есть сам процессор-исполнитель, ОЗУ, ПЗУ, порты ввода вывода, счетчики, АЦП, COM-порт и все это перевязано шинами данных, адреса и управления.

Вот как работает процессор (на этот механизм мы повлиять не можем он строго зашит и является личностью и характером процессора):

1. Пользователь включил питание.
2. Процессор ожил, но спит и видит сон: беспорядочно переключаются биты.
3. Процессор увидел, что питание достигло уровня >95% от стандартного.
4. Процессор пропустил тактовый генератор на схему исполнения команд, заодно инициализировал несколько системных регистров. Один из них адрес текущей точки выполнения команд.
5. Процессор начал выполнять команды, вытаскивая их по одной по порядку начиная с ячейки памяти с адресом ноль (или как там у него принято)
6. Наш процессор «**Advanced RISC Architecture**», что означает, что почти все команды выполняются за один такт частоты. Один такт - 1/16 000 000 секунды.
7. При выполнении одного такта почти параллельно делаются следующие операции:
    - Процессор выставляет адрес на шину адреса (для адресации 64 кБ необходимо 16 проводов в шине, т.е. шина адреса 16-ти битная)
    - Процессор выставляет управляющую команду на шину управления (шина управления 3-6 проводов: чтение/запись, запрос ОЗУ, запрос ПЗУ, запрос FLASH, запрос портов ввода вывода, прерывания). Команда например: «Память дай данные с адреса который выставлен на шине адреса».
    - Память дала данные на шину данных (8 проводов, т.е. 8-ми битная) и дернула за шину управления "данные готовы".
    - Процессор взял данные. В данных лежала команда для исполнения. К этой команде возможно нужны еще несколько байтов (по 8 бит считываний из памяти). Если это так, то процессор их дочитал.
    - Процессор выполнил команду (это могла быть команда сложения или записи или перехода это тоже целая серия работ с шинами данных, управления и адреса)
    - Процессор рассчитал адрес следующей команды и прочитал ее из памяти потом выполнил и так далее.

Так работает железо. Все достаточно однообразно. Внешне мы смотрим фильм или слушаем музыку, а в кочегарке кто-то выполняет одни и те же команды чтения, записи и сложения. Например, чтоб нарисовать точку на экране, процессор в видеопамять записывает код цвета точки, разбитый на спектральные составляющие красного, синего и зеленого по байту на каждый параметр. 3 байта нужно для зажигания одной точки, а зажечь надо 1280x1024 точек и перерисовывать их 50 раз в секунду. Чтобы сыграть мелодию на динамике, надо на катушку, оттягивающую мембрану динамика, подавать величину оттяга от 0-255 (1 байт) с частотой 44 000 раз в секунду. Все команды суть одно и тоже - это чтение запись и арифметика значений в ячейках памяти, а в это время пользователь видит ролик через интернет или общается с другой страной по Skype.

Создатели Си спрятали от нас всю рутину программирования на ассемблере и машинных кодах и предложили обобщенный интерфейс более приближенный к инженерному смыслу. Мы будем работать с «температурой и давлением», а не с движением множества хаотичных атомов. И все же, полностью полагаться на возможности Си мы не будем. Вот если бы мы программировали на intel i7 920, то конечно, все изыски Си в нашем распоряжении, но т.к. мы программируем для ATMega32, нам приходится искать золотую середину, что взять от Си, а что сделать самим. Мы слишком скованы ограниченной памятью и быстродействием. Мы возьмем от Си только самое необходимое.

Думая на языке Си, мы можем быть более свободны чем в ассемблере. Мы более произвольно можем располагать куски программы. Компилятор наведет порядок и самым оптимальным способом трансформирует наши мысли на Си в язык машинных кодов.

Создатели компилятора боролись за каждый сэкономленный байт при трансляции стандартных функций. Мы не смогли бы добиться такого же качества машинного кода, если бы писали его на ассемблере сами.

Компилятор оптимизирует код по быстродействию и/или по размеру. Не все версии компиляторов работают одинаково. Придется выбирать нужный проводя сравнения производительности.

Итак мы знаем, что программа - это последовательность действий для процессора. Программа должна предусмотреть все возможные ситуации в которые попадает наше ЗУ и должна содержать инструкции что делать процессору и как выпутаться с честью. А если пользователь отцепит аккумулятор во время зарядки?! А если пользователь закоротит зарядку?! А если пользователь забудет зарядку включенной и уедет в отпуск?! А если аккумулятор перегреется и сожжет квартиру?!

Программа должна предусмотреть все! Программа должна быть надежной! Программа должна быть удобной. Программа должна экономить энергию. Программа должна экономить время пользователя. Программа должна бережно обращаться с аккумуляторами. Программа должна быть не тормозной и быстро реагировать на команды пользователя. Программа должна четко отчитываться о проделанной работе. Программа должна быть лучше чем другие. Программа должна быть защищена от глупых и случайных действий пользователя.

Программа это своеобразное живое существо которое живет внутри ЗУ и управляет им. Программа это сложное переплетение маленьких простых кусочков. Переходы между ними могут быть сильно запутаны. Язык Си помогает нам найти удобную форму для записи наших оригинальных мыслей так, чтобы потом мы могли понять себя и чтобы нас поняли другие, поэтому надо быть консерватором в стиле программирования и Моцартом в идеях.

Для освоения языка Си необходимо узнать мнемонику языка (характерные слова и символы для записи программы). Все языки высокого уровня суть одно и тоже и отличаются только мнемоникой. Все языки программирования (до нынешних времен многоядерности) подразумевали один поток исполнения команд, как человек, работа, дом, работа, дом, магазин, если есть деньги - ресторан, отпуск, больница, работа, дом. Поток выполнения команд перескакивает то туда то сюда в зависимости от условий жизни, попадает в циклы работа дом и в конце концов смерть - окончание действия программы.

Программа металась туда сюда что-то делала, но смысла в своих действиях не видела. Смысл виден, если вырваться за пределы программы. Аккумулятор зарядился, был вставлен в самолет, который оторвался от земли. Также и со смыслом жизни человека.

Итак все программы - это условия, переходы и циклы, а между ними вызовы функций. Вот и все программирование. Итак классические операторы условия и циклов:

```
Классическое условие
if(условие)
  {
  внутренность этих скобок выполняется однократно если условие выполнилось
  }
else          // допускается отсутствие else
  {
  внутренность этих скобок выполняется однократно если условие НЕ выполнилось
  }
далее программа выполняется здесь

Классический цикл wihle
while(условие)
  {
  внутренность этих скобок выполняется до тех пор пока справедливо условие
  }
далее программа выполняется здесь

Классический цикл do wihle (минимум однократное выполнение тела)
do{
  внутренность этих скобок выполняется 1 раз и до тех пор пока справедливо
  условие
  }while(условие);
далее программа выполняется здесь

Классический цикл for
for(i=0; i<100; i=i+1)
  {
  внутренность этих скобок выполнится 100 раз
  первый раз i=0
  второй раз i=1
  последний раз i=99
  }
далее программа выполняется здесь

Классический переключатель
switch(x)
  {
  case 0:     // в случае если x=0
    тело программы
  case 1:     // в случае если x=1
    тело программы
  case 67:    // в случае если x=67
    тело программы
  default:    // во всех остальных случаях, не описанных выше
    тело программы
  }
далее программа выполняется здесь
```

Во всех циклах и **switch** допускается использование оператора **break;** Этот оператор досрочно завершает работу цикла. В циклах while и for можно использовать оператор **continue;** Этот оператор завершает выполнение текущего прохода цикла и переводит программу на выполнение следующего.

## Урок №5 (Переменные)

Теперь, когда приблизительно ясно что такое процессор, что такое язык, что такое ход выполнения программы, т.е. есть общий взгляд на картину, начинаем приближаться и рассматривать ее более детально. Где-то выше мы отмечали, что всему в мире можно присвоить код и этот код можно хранить в памяти компьютера в виде напряжений в ячейках памяти, которые устроены как конденсатор (заряжен/не заряжен) или как триггер защелка из

6-ти транзисторов (который имеет 2 устойчивых состояния) или это домен (маленький магнит) в магнитном слое на винчестере/дискетке/ленте или это изменение свойств вещества компакт диска или это дырка в перфокарте как это было на заре цивилизации.

Итак всё вышеперечисленное - это 1 бит информации - да/нет, инь/янь, 0в/+5в, дырка/недырка, север/юг магнита, мужчина/женщина, нолик/единичка, день/ночь, порядок/хаос. Всего 2 состояния, поэтому и двоичная система исчисления.

В двоичной системе всего две цифры 0 и 1. В двоичной системе нет цифры 2, но есть число 2 и оно записывается так: 10.

В десятичной системе исчисления 10 цифр: 0,1,2,3,4,5,6,7,8,9 и тоже нету цифры 10, но есть число 10 и записывается оно: 10.

В шестнадцатеричной системе исчисления 16 цифр: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f и тоже нет цифры 16, но есть число 16 и записывается оно: 10

Теперь сделаем важное замечание: при попытке записать число в разных системах исчисления, **количество предметов не меняется, меняется способ записи.** Меняется вид, но смысл остается тот же. В тексте программы мы будем записывать абсолютный смысл числа предметов разными цифрами, имея ввиду одно и тоже количество.

Если вам надо перевести какое то число из одного вида исчисления в другой, воспользуйтесь калькулятором Windows (в настройках указать вид-инженерный).

Если мы говорим про ноги процессора, то нам удобнее двоичный вид.

Если мы говорим про целые числа предметов, то нам удобнее десятичный вид.

Если мы говорим про большие адреса или большие двоичные числа, то удобнее 16-теричный вид.

16-теричная и двоичная система очень похожи и удобны в преобразовании из одной в другую. Двоичная система очень громоздкая, 16-теричная — компактная.

Исторически сложилось что первые нормальные процессоры были 8-битными. Это значит что в одной ячейке информации параллельно хранились 8 битов информации и это значит, что шина данных между процессором и памятью состояла из 8ми проводов. Запрашивая данные из ячейки памяти процессор получал одновременно 8 битов. Современные компьютеры доросли до 64 битной шины данных процессоров, а в графических картах уже используются шины данных до 256 бит.

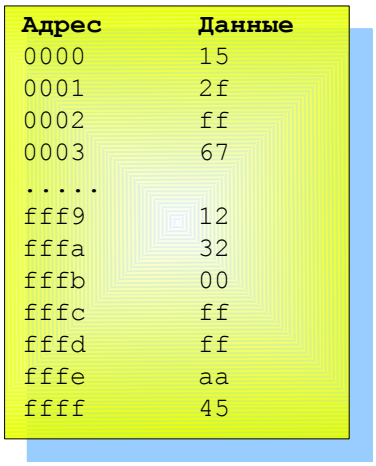
8 битов информации могут хранить в себе число от 00000000 до 11111111 или в десятичной от 0 до 255 или в шестнадцатеричной от 00 до ff.

Процессор ATMega32 это 8 битный процессор, значит все вышесказанное относится к нам.

С данными разобрались, теперь адрес. Если данные это содержимое ящичка, то адрес это номер ящичка по порядку. И то и другое есть число. И то и другое записывается в 2,10,16-теричном исчислении. Данные хранятся в ящичках — в ячейках памяти, адрес нигде не хранится. Адрес — это последовательные номера ячеек. Ячейки находятся в микросхеме памяти и физически располагаются по порядку. Зная адрес мы можем найти ту самую ячейку. Но адрес можно превратить в данные, записав его в ячейку.

Необходимо четко понимать: в компьютере есть только два вида чисел: данные и адрес. Все остальные слова, придуманные международным сообществом (ссылка, метка, идентификатор, указатель, дата, код, цвет и т.д. ...) - это все либо данные либо адрес. Когда я себе представляю мысленно память компьютера, я вижу последовательность ячеек. У каждой

ячейки есть свой адрес (номер ячейки), а внутри ячейки хранятся данные.




Все данные хранятся в ячейках памяти компьютера, каждая ячейка имеет адрес и в каждой ячейке лежит один байт.

Хотите писать программу с календарными датами, числами с плавающей точкой, с большими целыми числами - выкручивайтесь как хотите. Это ваши проблемы. Компьютер устроен так и не иначе.

Вот мы и будем выкручиваться как хотим, но в соответствии с международными стандартами.

Для того чтобы компилятор понимал с какими данными мы работаем в нашей программе, мы должны явно указать типы всех наших данных. Четко зная тип данных, компилятор выделит строго определенные ячейки данных и будет в них хранить наши данные-переменные.

Некоторые компиляторы неСи не требуют описания переменных, тогда они сами догадываются о типе из текста программы или во время работы программы меняют тип, но это излишние затраты памяти и быстродействия, а это произвол и нам это не подходит.

Наш компилятор предсказуем, отсебятину не несет, делает строго то что мы просим и требует от нас исчерпывающей однозначно понимаемой информации. В противном случае компилятор сигнализирует о неполноте программы.

## Урок №6 (Типы переменных)

| Название | Размер | Значение | Комментарий |
|---|---|---|---|
| bool | 1 bit | 0 или 1 | Правда или ложь. Используется для логических конструкций. |
| char | 1 byte = 8 bit | от -128 до +127 | маленькое целое число или код буквы . Классический вариант. |
| unsigned char | 1 byte = 8 bit | от 0 до 255 | маленькое целое беззнаковое число или код буквы. Классический вариант. |
| BYTE | 1 byte = 8 bit | от 0 до 255 | Все то же самое что и в предыдущей строке только записывать удобнее. |
| int | 2 byte = 16 bit | от -32768 до +32767 | Целое знаковое число. |
| unsigned int | 2 byte = 16 bit | от 0 до 65536 | Целое без знаковое число. |

| | | | |
|---|---|---|---|
| WORD | 2 byte = 16 bit | от 0 до 65536 | Все то же самое что и в предыдущей строке только записывать удобнее. |
| short или short int | 1 byte = 8 bit | от -128 до +127 | маленькое целое число или код буквы . Классический вариант. |
| long или long int | 4 byte = 32 bit | от -2147483648 до +2147483647 | Целое знаковое число. |
| float | 4 byte = 32 bit | | Числа с плавающей точкой |
| double | 8 byte = 64 bit | | Числа с плавающей точкой для точных вычислений |
| long double | 10 byte = 80 bit | | Числа с плавающей точкой для очень точных вычислений |

В таблице выделены те типы которые будут использованы в программе. Чем меньше используем типов, тем больше экономим места в памяти процессора.

ВНИМАНИЕ! Язык Си, если не указано в настройках, различает большие и маленькие буквы. Т.е. написание букв большими или маленькими важно и отслеживается.

**Пример использования в Си:**

```
BYTE x=7;
unsigned char y=0x45, z=b00010010;
```

В этой строке мы сказали компилятору выделить где-нибудь где удобно в памяти 3 ячейки. В одной ячейке будет храниться переменная "x" в другой "y" в третей "z". Компилятору абсолютно все равно как вы назовете переменные, для него важно, что везде где встретится x в программе он будет вспоминать о том месте, где он выделил память, и будет оттуда брать саму переменную или записывать ее туда. Название переменной не попадет в текст машинного кода, значит длинна названия переменной никак не повлияет на экономию памяти машинного кода.

Переменные могут называться только латинскими буквами, цифрами и символом «_». Длина названия переменной до 32 знаков причем первая обязательно должна быть буква или символ«_».

Когда компилятор выделяет память - это значит, что у себя он где-то просто запомнил адрес который он выделил, а с памятью непосредственно он ничего делать не будет. Слово "выделил" ничего такого волшебного не подразумевает. Когда мы говорим про переменные, мы всегда подразумеваем, что они хранятся в ОЗУ, в той самой памяти, которая быстрая и все забывает при выключении питания.

При описании переменной мы поставили знак "=", этим мы сказали компилятору вписать в программу на машинных кодах небольшой код по инициализации переменных. Это значит, что после того как ЗУ включится, первым делом программа на машинных кодах впишет в ячейку с адресом переменной (которую мы называли x) число 7, потому что ОЗУ при включении содержит в себе непредсказуемые данные.

В переменную "y" мы попросили записать 0x45 (таким образом мы сказали компилятору, что 45 это 16-теричное число).

В переменную "z" мы попросили записать b00010010 - двоичное число. Мне именно необходимо двоичное число, чтобы видеть на какой ноге будет +5в. Каждый знак в двоичной записи соответствует ноге ATMega32 - это очень удобно (если речь идет о портах ввода вывода A,B,C,D).

Разные компиляторы по разному требуют запись двоичного числа. Наш компилятор

никак не хотел принимать от меня двоичные числа и я обманул его, придумал свои двоичные числа (b00010010) и подменил их командой **#define** на те числа которые понятны компилятору. 256 таких команд для всех двоичных чисел размером в один байт я сложил в файл **bin.h** который включил в основной файл **main.cpp** командой **#include "bin.h".** Не смотря на то, что файл большой, на машинном коде он никак не отразится, т.к. этот файл всего лишь помогает трансформировать бинарные числа из текста программы (удобные для читаемости программы) в нормальные числа понятные компилятору.

Переменные типа **BYTE** будем использовать для маленьких целых для организации циклов, для работы непосредственно с памятью, для хранения всяческих маленьких настроек ЗУ.

**int** - это 2 байта в нашем компиляторе. В памяти хранится сначала младший байт потом старший байт. Когда мы говорим об адресе в памяти для переменной **int**, то говорим об адресе младшего байта, т.е. первого из двух. Два байта может вместить в себя без знаковое число от b0000000000000000 до b1111111111111111 или от 0 до 65535 или от 0x0000 до 0xffff. Думаю уже понятно что есть что. А если число знаковое, то вот что придумали программисты: они как бы перекинули половину диапазона в отрицательную часть числа. Получилось, что знаковый int может принимать значения от -32768 до 32767 это ровно такое же количество чисел как и было.

Никто не пользуется отрицательными двоичными и шестнадцатеричными числами это может привести к путанице. Двоичные и 16-теричные используются в контексте реальных адресов и данных и непонятно что такое отрицательный адрес. Но заметим одну важную особенность: у отрицательного числа самый старший бит 1 (это как бы знак минус). Больше на эту тему распространяться не буду. Хотите точнее узнать - играйтесь с инженерным калькулятором Windows.

**int** - это знаковое целое (-32768 до 32767)

**unsigned int** - это без знаковое целое (0-65535)

**unsigned int** писать очень долго поэтому сделаем **#define WORD unsigned int** и будем везде писать **WORD**

Этот тип переменных мы будем использовать для счетчиков циклов, для средних целых, для ШИМ счетчиков, для контрольных сумм, для АЦП суммирования.

Заметьте уже второй тип переменных строго ограничен в своих значениях. С одной стороны, мы точно знаем что потратили только 2 байта при использовании переменной, а мы будем экономить каждый байт, но, с другой стороны, мы получили головную боль по переполнению переменных и жесткому отслеживанию рамок. Например если к WORD(65535) добавить 1 то получится 0. А если от WORD(0) отнять 5, то получится 65531 и компилятор вам не поможет в этом вопросе, т.к. вычисления производятся уже после того как программа с машинными кодами уже во всю заряжает и разряжает.

По личному опыту, будьте внимательны, очень часто совершаются ошибки с переполнением или со сравнением разных типов. Эти ошибки очень трудно найти, только с отладчиком, но какой отладчик в ЗУ при зарядке? В принципе можно и отладчик, но это такие сложности с дополнительным железом. Легче писать правильно и иметь опыт. Тоже самое эмуляторы и симуляторы - очень неудобны, отнимают много времени. Мне кажется, что наиболее эффективны 3 способа нахождения ошибки: внимательно проверить программу, спросить специалиста или исправить программу с целью поставить опыт прямо на ЗУ с выводом необходимой информации на экран ЗУ. К сожалению, это поможет, если вы уже переправились и захватили плацдарм в виде хотя бы одной отзывчивой прошивки и от нее можете танцевать.

Все что сказано про **int** (integer - целое число без дробной части) справедливо и для других двух типов: **short** (короткий 1 байт) и **long** (длинный 4 байта).

**unsigned short** это тоже самое что и **BYTE**, за одним маленьким исключением: некоторые стандартные функции WinAVR по бюрократски требуют именно short, а не BYTE.

**long** - это очень много - это можно считать сотые доли секунды в течение всего дня

и long не переполнится.

Ну и наконец **float** это число с плавающей точкой занимает 4 байта. Как хранится в памяти мне не известно да и не надо знать. Все точные вычисления до 4 знака после запятой будем делать в переменных типа **float** (токи, напряжения, сопротивления, балансировочные коэффициенты, интегралы-суммы, дифференциалы-наклоны). Это самые медленные операции. Одна графическая карта ATI или nVidea может пережевать миллионы операций с плавающей точкой, а ATMega32 не более 1000 операций в секунду и это сильно тормозит все остальные операции ЗУ. Поэтому перед тем как программировать надо очень хорошо подумать как сократить количество вычислений как упростить все формулы, а что можно вычислить заранее и ввести в формулу в качестве коэффициента.

Еще одно правило: необходимо таким образом составлять формулу, чтобы компилятор на каждом этапе вычислений находился в среднем диапазоне чисел. Это повысит точность вычислений. На каждом этапе вычислений точность падает до величины округления. Поэтому старайтесь чтобы каждое вычисление внутри формулы приводило к результату от 0.001 до 1000 (по модулю) и хранилось в этом же диапазоне и только в самом конце (перед отображением) приводилось к истинному порядку величины. Такие простые правила позволят не потерять точность на стадии вычислений.

Последний тип который нам понадобится - **bool** от слова boolean - тип логической переменной, которая может принимать два значения true (правда) и false (ложь), т.е. это и есть один бит информации. Хранить один бит информации в памяти где все хранится по 8 бит это у компилятора вызывает трудности: либо наплевать на экономию и хранить бит как байт (0-ложь, >0-правда) либо постараться с запоминанием в каком байте какой бит соответствует нашей переменной.

Мы то точно будем бороться с этой темой, т.к. мы жадные, а bool переменных (флагов) у нас много. Но об этом потом.

В заключение скажем, что меньшая банка помещается в большую. Например **long = int** компилятор спокойно преобразовывает, дополняя нулями все незанятое место. А вот в обратную сторону могут происходить чудеса. При преобразовании int=long или BYTE=WORD, присваивается младшая байтовая часть правой стороны, которая занимает полностью присваиваемую переменную.

## Урок №7 (операторы)

До сих пор я много говорил про команды процессора (управляющие ноги процессора, мостов, памяти, устройств ввода/вывода), говорил про команды машинных кодов, говорил про функции (подпрограммы и программы).

При упоминании про команды сразу представляется, что есть некий исполнитель и есть некая последовательность действий (команда), которую надо исполнить.

Как видите, мы утопаем в количестве информации про которую надо говорить и употребление слова команда приводит в замешательство: в каком смысле оно используется, поэтому при описании языка Си, имея ввиду команды, мы будем использовать слово ОПЕРАТОР.

Оператор - это то, что должен исполнить процессор при выполнении программы.

Оператор это как бы логическая единица выполнения.

Оператор это одно выражение, заканчивающееся ";".

Оператор может быть сложным, состоящим из множества функций.

Оператор может состоять из одной функции. Функция и оператор это почти одно и тоже.

Программа Си - это последовательность операторов. В ассемблере программа это последовательность машинных команд. В Си оператор - это более емкое понятие более приближенное к жизни. В ассемблере мы должны помнить из чего состоит переменная и

работать с каждой ее частью. В Си мы работаем целиком с переменной как с единой сущностью, не переживая как там все устроено.

Операторы - это тело программы.

Оператор - это фрагмент программы. При написании программы операторы могут располагаться в одной строчке или каждый оператор на новой строчке.

Оператором может называться как все выражение от ";" до ";" так и самое главное слово или знак, отражающий смысл оператора.

**Например:**

```
x=47*y;              // Здесь оператор "="
goto qqq;            // Оператор перехода goto
for(int i=0; i<45; i++) // Оператор цикла for
break;               // Оператор выхода из цикла
if(x<y)e=4;          // Оператор сравнения if...
```

## Урок №8 (Первая настоящая программа на Си для ATMega32)

```
#include <avr/io.h>                    // Общая муть
#include <avr/iom32.h>                 // Мега 32 стандартные описания
#include <avr/wdt.h>                   // Стандартные описатели собаки
#include <util/delay.h>                // Стандартные описатели задержек
#include "bit.h"                       // Обращение к битам портов по названию

// Главная программа запускается по ресету
int main(void)
{
// Это магические слова, чтобы ЗУ себя не спалило
cli();           // Запрет всех прерываний
WDTCR=15;        // Инициализация сторожевого таймера специально 2 раза
WDTCR=15;        // иначе не сработает (защита от случайной порчи собаки)
PORTA=0;         // Порт A нули 0000 0000
PORTB=0x1c;      // Порт B нули 0001 1100 кроме PB2 PB3 PB4 (клава на +)
PORTC=0x80;      // В порту C все нули 1000 0000 кроме PC7 (клава на +)
PORTD=0x02;      // В порту D все нули 0000 0010 кроме PD1 (подтяжка TXD на +)
DDRA=0;          // Задаем направление работы ног порта A 0000 0000
DDRB=0x43;       // Задаем направление работы ног порта B 0100 0011
DDRC=0x7f;       // Задаем направление работы ног порта C 0111 1111
DDRD=0xfe;       // Задаем направление работы ног порта D 1111 1110

// Этот цикл выполняется бесконечно. То что нам надо!
while(true)
  {
  wdt_reset();                    // Сброс собаки
  _delay_us(10000);               // Задержка 0.01 секунды
  PORTD_Bit7=1;                   // Подать напряжение на динамик
  _delay_us(10000);               // Задержка 0.01 секунды
  PORTD_Bit7=0;                   // Убрать напряжение с динамика
  }
// Хотя сюда программа никогда не доберется, мы обязаны закрыть все скобки и
соблюсти все правила хорошего тона.
}
// Конец главной программы
```

После запуска этой программы мы услышим тарахтение динамика.

Для того чтобы ехать быстро, приходится очень долго запрягать. Вот и в нашем случае, мы будем ставить эксперименты на ЗУ и, чтобы его не спалить, приходится писать лишний код. А также в начале программы мы подключили очень много файлов описателей. Просто не обращайте на них внимание. В зависимости от того какие вы используете

стандартные функции и операторы в своей программе, необходимо подключать соответствующие файлы описателей. Этот вопрос решается экспериментально или читая help на операторы и функции.

Самое интересное в этой программе, то из-за чего все было затеяно, это внутренности оператора while.

ATMega32 будет бесконечно выполнять все операторы внутри цикла, т.к. условие **while(true)** всегда истинно, значит дойдя до нижней скобки цикла "}" и проверив условие (true), программа начнет цикл заново от верхней скобки "{".

Внутри цикла выполняются последовательно оператор **wdt_reset();** - сброс собаки, **_delay_us(10000);** ничего не делать (выполнять пустой код, растрачивая время) в течение 10000 микросекунд, т.е. 0.01 секунды, потом установить на 16 ноге процессора 5в **PORTD_Bit7=1;**, потом опять подождать и сбросить **PORTD_Bit7=0;**. Т.е. Оттянуть и отпустить мембрану пищалки.

## Урок №9 (Вычисления выражений)

Особенности вычисления длинных выражений.

Рассмотрим пример:

```
x= (...) / ((...) + ((...) * (...)));
```

Как бы, вы, вычисляли данное выражение? Компилятор это делает так же как делали бы вы. Компилятор находит самые внутренние скобки и в них два операнда (т.е. числа слева и справа от знака операции (слагаемых, вычитаемых, умножаемых, делимых) далее выполняет над ними операцию и получает результат. Результат сохраняется во временной переменной. Таким образом выражение упрощается вместо двух операндов и операции осталось вычисленное значение - временная переменная. Далее компилятор опять ищет самые внутренние скобки и т.д. В конце концов от всего выражения остается одно значение — общий результат, лежащий во временной переменной, который вписывается в память туда, где у нас должна располагаться переменная "x".

Само вписывание и есть оператор присваивания. НО возникает два вопроса про детали вычислений:

1. Если компилятор нашел внутренние скобки, а там a+b*c, то сначала складывать или умножать? То чему нас учили в школе в принципе компилятору известно, но жизнь и все возможные ситуации не укладываются в школьные представления, потому что есть и другие операции кроме +-/* и определить порядок вычислений на все случаи жизни бывает очень трудно, да и запоминать тот порядок который принят у компилятора трудно. Выход есть! Самый высокий приоритет у скобок. Если вы скобками подскажете компилятору порядок, то однозначно будете уверены, что все будет сделано правильно. Чем больше скобок, тем лучше. Скобки не отразятся на длине машинного кода. Совсем все скобками заполнять не надо, только там где вы не уверены.
2. А как компилятор будет складывать 5 груш и 4 яблока? Или, например, сколько будет 5/4? Что происходит с переменными разных типов при вычислениях? Напоминаю, что если в вашей программе где либо встретится выражение 5/4, значит ваша программа неэкономная. Все что можно вычислить заранее, надо вычислить и вписать готовое в программу. В программе должны быть только переменные и не сокращаемые коэффициенты. Я пишу 5/4 как пример, чтобы сразу было видно, что тип числа целое/целое. Компилятор выполняет только те операции которые у него описаны внутри его создателями, все непонятки компилятор приводит к понятным типам "по умолчанию", т.е. как сделали авторы компилятора по рекомендациям международного сообщества Си. Если операнды (слагаемые...) разных типов, то надо их привести к одному типу, а как складывать одинаковые типы ясно.

Существует список из 20 правил по которым неодинаковые типы приводятся к одинаковым. Приводить эти правила я не буду, потому что разные компиляторы имеют некоторые отличия, но в целом можно сказать, что более простой тип из двух приводится к более сложному и операция выполняется над двумя сложными.

И опять есть выход, который позволит нам точно быть уверенными, что компилятор все сделает правильно: мы можем явно подсказать компилятору, что делать с типами. Например int(x) - превратит x в целое число, просто отбросит дробную часть без округления, а остатки как может запихает в целое число.

После вычисления правой части выражения, результат оказывается во временной переменной (без имени), которая может отличаться по типу от переменной слева - x. Перед записью в x временная переменная тоже трансформирует свой тип в тип переменной x.

А как понимать выражение "x=x+1"; ? **Это не уравнение**. Это запись обычного выражения Си. И вообще, в программировании нет никаких уравнений. За уравнениями обращайтесь в МАТКАД. Все выражения в Си - это порядок вычислений. Порядок "строго" задан все данные в наличии имеются, никаких колдований с решениями уравнений в программе нет. Вы уже все наколдовали заранее и в программу поместили готовые формулы расчета корней.

Поэтому, компилятор, увидев x=x+1, делает следующее: достает из адреса где лежит x содержимое (значение x) добавляет 1 и кладет это значение назад в x.

Кому может понадобиться добавлять 1 к переменной? Очень часто используется такое выражение, например перебирая массив данных, работая с каждым элементом, мы указываем номер элемента или меняем переменную цикла, чтобы цикл выполнился ровно 100 раз. Для упрощения записи, только в Си, придумали записывать x=x+1 несколько иначе:

```
x=x+1;
x++;
++x;
x+=1;
```

Все эти выражения означают одно и тоже x=x+1;

Например если вам надо выполнить цикл 100 раз, то пишем:

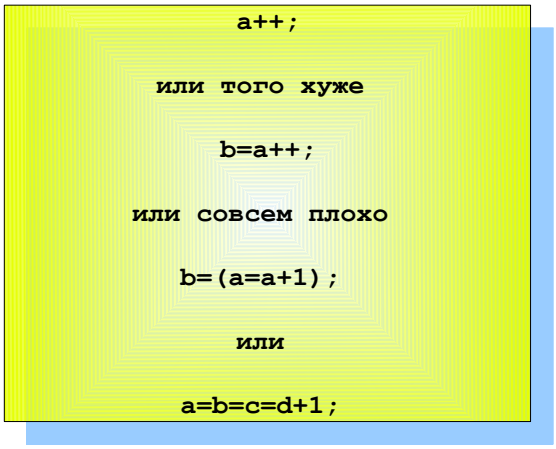
```
for(int x=0; x<100; x++)
  {
  Внутренности скобок выполнятся 100 раз
  и при каждом прохождении цикла
  переменная x будет содержать
  число от 0 до 99 по порядку
  }
```

А если надо добавить 2, то:

```
x=x+2;
x++; x++;
++x; x++;
x+=2;
```

## Урок №10 (философия Си)

Создатели языка Си очень экстравагантные люди. Это видно по тем конструкциям, которые они придумали. Например чего стоят:




Таких конструкций нет ни в одном языке программирования. Подобного рода конструкции осуществляют присваивание справа налево, т.е. сначала осуществляется расчет выражения и присвоение справа, а потом все левее и левее.

Конструкция b=a++; означает, что в переменную "b" будет скопировано содержимое переменной "a", а потом "a" увеличится на 1.

Конструкция b=++a; означает, что сначала "a" увеличится на 1, а потом увеличенное содержимое "a" скопируется в "b".

На этом создатели Си не остановились. В каком то ветхом году уже был придуман язык Си++ (поэтому названия файлов с кодом *.cpp) в котором была совершена революция в обобщении типов (или структур) переменных и операций над ними.

Было введено понятие "класса" и "объектно ориентированного программирования", которое потом было подхвачено другими языками.

Обо всем можно говорить просто и также просто мы скажем о классах, но сначала подытожим наши знания о Си и языках программирования.

Итак есть железяка - ЗУ или винчестер или флэшка или комп или электронный замок с таблетками или ЖПС навигатор или любая другая железка с процессором внутри. Сама по себе железка ничего делать не умеет, в ней заложена возможность. Человек создан по образу и подобию Бога. Но вот чем он себя наполнит? Пивом?

Без программы мозг железки пуст. Чтобы она ожила, необходима программа. Программа формирует поведение устройства. Устройство либо спит и ждет команды человека, либо дежурит выполняя монотонные действия, ожидая сигналов от датчиков, кнопок клавиатуры, сигналов из сети и т.д.

Потом появляется человек, передает команды, устройство их выполняет и возвращает результат в виде изображения, звуков или других воздействий на наши органы чувств. И нам кажется, что мы общаемся с искусственным разумом, который реагирует на наши команды и выдает нам результаты. А на самом деле, мы общаемся с тенью программиста, который когда-то сформулировал все модели поведения программы.

Для реализации всех возможных моделей программист мысленно как бы проделал все возможные жизненные ситуации и подсказал программе как надо вести себя. При этом требуемая информация лежала в массивах памяти и переменных, преобразовывалась и записывалась в другие переменные и воздействовала на внешний мир через порты ввода вывода. Сама работа программы - это продвижение по ее тексту, как ребенок водит пальцем по тексту книги, и в том месте где палец, те команды непосредственно в данный момент выполняются.

Текст программы ветвится логическими конструкциями в зависимости от

содержимого переменных, но в целом ходит кругами, пока устройство не отключат от питания.

Старый способ программирования (до "классов") подразумевал, что есть строго ограниченное количество типов переменных и операций над ними. Например целые числа можно складывать, делить, умножать, вычитать и т.д. при этом получаются тоже целые числа.

Все многообразие мира описывается через эти переменные которыми задаются свойства всех объектов мира. Например дерево имеет столько то веток, оно столько то весит, оно имеет коричневый ствол и зеленую листву с рисунком из жилок. Дерево растет со скоростью х и потребляет кислород и вырабатывает его же. Весь образ дерева хранится в памяти программиста и он вписывает в программу о дереве информацию в отдельные переменные.

В языке Си++ можно создать класс Деревья (как тип объектов) непосредственно объект Дерево (как саму переменную), описать все операции которые можно выполнять с деревом (как сложение вычитание и т.д.) Описать события, которые случаются с деревом (посадили, выросло до 3 метров, спилили, сожгли), описать свойства дерева (рост, вес, цвет и т.д.). В целом ничего не изменилось: все свойства дерева так и хранятся в переменных, все функции-методы необходимо отдельно записать. Просто в С++ прописан механизм как собрать воедино все связанное с объектом и удобно оперировать им.

Таким образом существует новый подход к описанию мира и созданию образов в программировании, более близкий к тому как думает человек, дающий новые возможности и удобства.

В языке Си++ есть еще одна возможность - "наследование". Чтобы создать новый образ "Береза" не надо заново описывать все, что присуще березам, мне достаточно сказать компилятору, что березы это наследник деревьев и все, что присуще деревьям, присуще березам, но есть новые свойства, методы и события, которые присущи только березам.

Программы, написанные на Си++ с использованием классов, имеют несколько другой облик. Они как набор подпрограмм, которые выполняются при наступлении событий всех объектов, описанных в программе. Например есть лес, состоящий из уникальных деревьев-объектов. Вот как выглядит программа на Си++

1. Программа спит и ждет событий. Если это многозадачная система, то данное приложение ресурсы не использует.
2. Эта подпрограмма выполняется если объект создается заново. Выделяется динамическая память, инициализируются переменные свойств. Запускаются какие-нибудь методы: полив саженца.
3. Эта подпрограмма описывает, что делать при наступлении события С. Например если наступает засуха, то полив.
4. Эта подпрограмма выполняется в час Ч. Сбор урожая.
5. Эта подпрограмма совершает необходимые действия при удалении объекта. Освобождение памяти.

Объектно ориентированное программирование - это как бы сборник обработчиков всех возможных событий. А внутри этих событий выполняются методы/операции над объектами и их свойствами (создаются, уничтожаются, перекрашиваются, увеличиваются и уменьшаются). Чтобы все это многообразие однозначно описать для компилятора, понадобилось некоторое количество дополнительных служебных слов и знаков.

А теперь о грустном: Это конечно все хорошо, но не годится для маленького процессора. А все эти возможности можно реализовать и в простом Си, но с несколько более тяжелой грамматикой и орфографией и большей умственной нагрузкой на программиста.

Нам придется отказаться от классов из экономии, хотя как было бы удобно программировать на Си++ !!! Скорее всего я заблуждаюсь. Время покажет.

Например так могла бы выглядеть наша программа на Си++:

```
#include <avr.h>
…
#include «Tlcd.h»
#include «Tchennel.h»

void OnTimeToPrint(void)
{
LCD->Restart();
LCD->Clear();
LCD->SetCursor(3, 7);
LCD->Print("Welcome!");
LCD->LoadNewSymbol(0x04, 14, 45, 44, 1, 3, 6, 7, 1);
LCD->SetCursor(1, 0);
LCD->Print(0x04);
}

void OnTestChannel(TChannel Ch)
{
Vst=Ch->GetStaticVoltage();
Ist=Ch->GetCurrent();
}
```

В данном примере к объекту LCD применяются методы, а в результате программа посылает на LCD различные данные. Объектно ориентированное программирование почти добралось до идеала программирования, где в свободной форме предъявляется техническое задание, а на выходе получается готовая программа.

Но, за то С++ прекрасно работает на большом компьютере. Сейчас буду говорить как написан Windows. Тоже очень коротко. Просто для ощущения простоты главной идеи. Сложности наступают в деталях, а главная идея всегда очень простая.

Так вот весь Windows написан на Си++ и именно с применением классов. Все окна, все поля ввода, все диски, принтеры все устройства и драйвера и вообще всё в Windows это классы. Почти все классы связаны между собой родством, т.е. многие классы являются наследниками других классов, т.е. имеют общее в своих свойствах, методах, событиях. У многих объектов Windows есть место где они рисуются на экране (окно), а значит есть размеры этого окна, положение и цвет.

Так вот в Windows есть мощный механизм взаимодействия между всеми этими объектами — это сообщения. Сообщения — это макро обобщение механизма прерываний. Сообщения переносят информацию (воздействие) от объекта к объекту, как фотоны или гравитоны. В физике тоже все построено на частицах (объектах) и силах взаимодействия. Как же все похоже устроено во всем мире!!!

Так вот, все эти объекты в Windows перевязаны сообщениями. Сама операционная система это огромный генератор сообщений, все что вы делаете с клавиатурой мышкой и дисками и сетями, все это порождает тысячи сообщений от операционной системы всем открытым окнам и самому верхнему окну-программе. Сообщения принимаются окном и обрабатываются. Сообщение - это «классовое событие» для окна. Система сообщила окну, что драйвер клавиатуры получил код нажатой кнопки. Окно вызвало обработчик события "кнопка нажата" и послала сообщение строке ввода там где сейчас моргает курсор, а строка приняла сообщение и послала сообщение драйверу видеокарты отображай кусок измененного экрана. Попутно все это было залогировано, попутно антивирус проверил нет ли вируса внутри кода кнопки (это конечно шутка, но если бы это был диск или сетка то это так). Вот мы и объяснили как работает Windows!!!

Для того чтобы плюхнуться в океан Windows нужно иметь самое общее

представление об устройстве и тогда становится значительно легче, по крайней мере, мне.

Больше про классы говорить не будем. Этот маленький экскурс показал нам идеал программирования на сегодняшний день к которому надо стремиться или не надо.

Наша узкая специфическая задача — создать маленькое программное устройство, маленького робота, в котором все экономно и оптимально, ничего лишнего и максимальная производительность.

Теперь сконцентрируемся например на ЖКИ и на использовании прерываний. Иметь аппарат для удобной работы с ЖКИ это пол дела при написании программы для электронного устройства с ЖК экраном.

## Урок №11 (прерывания)

Прерывания - это еще один удобный способ разветвить нашу программу в зависимости от событий которые происходят с ЗУ.

Где то ранее я утверждал, что при включении процессор начинает читать программу машинных кодов начиная с адреса 0. В процессоре ATMega32 все устроено несколько сложнее.

В начале памяти располагаются "ВЕКТОРА ПРЕРЫВАНИЙ" приблизительно 15-20 векторов по 2 байта на вектор. Каждый вектор - это адрес программы для обработки прерывания, т.е. адрес программы которая будет запущена при наступлении прерывания. Точно не уверен (для нас это не важно): чем ближе вектор к началу, тем выше статус прерывания, т.е. при наступлении двух прерываний одновременно, первым выполняется то прерывание, чей вектор ближе к началу.

Включение компьютера или RESET - это тоже прерывание которое имеет свой вектор. Компилятор сам впишет адрес в соответствующий вектор и расположит главную программу **main()** по указанному адресу.

В нашей программе на Си мы просто сообщаем компилятору, что мы задействовали то или иное прерывание и пишем подпрограмму для обработки. Компилятор сам впишет адреса всех подпрограмм в таблицу векторов прерываний.

Программа прерывания выполняется неожиданно при наступлении прерывания. Основная программа выполняла сложнейшие вычисления, храня данные как в ящиках стола, так и в карманах и в руках, но тут кто то приходит и говорит: бросай все ты мне поможешь чистить канализацию, переоденься. Процессор все распихивает по углам, набирает новые данные из вашей подпрограммы обработки прерывания и выполняет ее. Потом возвращается и восстанавливает естественный ход работы.

Прерывание было придумано, чтобы зафиксировать случившееся неожиданное и важное событие, поэтому программы обработки прерывания должны быть максимально маленькие и работать с минимальным количеством памяти и переменных, чтобы сильно не отвлекать процессор от основной главной задачи.

Хотя, бывает по-разному. Некоторые прерывания просят, вообще, все забыть и начать заново. После окончания обработки прерывания, процессор восстанавливает свое душевное и материальное состояние и продолжает, как ни в чем не бывало, работать над главной задачей всей его жизни.

Для того чтобы быстро распихать все свои данные перед вызовом прерывания и восстановиться после, процессор использует СТЕК. Стек вообще используют многие. При вызове каждой подпрограммы используется стек. Чем глубже вызовы подпрограммы из подпрограммы из подпрограммы из подпрограммы тем больше используется и растет стек.

Стек - это память где то в ОЗУ (обычно в конце) где ее выделит компилятор. Стек работает по правилу: то что засунули последним, то вытащится первым. Существует всего две команды ассемблера работы со стеком: засунуть в стек (push) и вытащить из стека (pop). Зная порядок как процессор запихивал в стек, процессор знает, что он оттуда достанет. Стек

нужен для экономии быстродействия. В стек как в карман засовываются адреса и данные. Если стек начинается в конце памяти, то при заполнении, он растет в сторону начала памяти. Стек может переполняться по мере использования, он может разрастись и залезть на область, где хранятся переменные, и тогда будет непредсказуемое поведение переменных и стека и все повиснет и понадобится собака.

Рас уж начали говорить про стек, немного поговорим про компилятор, чтобы иметь общее представление и знать ,что от него ждать.

## Урок №12 (компилятор)

В предыдущем тексте очень много раз упоминалось слово компилятор. Не побоюсь повторения и еще раз опишем его:

1. Компилятор это обычная программа, которая преобразует мысли программиста, сформулированные на каком нибудь языке в программу на машинных кодах, понятную процессору. Далее программа прошивается в память процессора (такое бывает только в SoC http://ru.wikipedia.org/wiki/SoC как наш ATMega32 — система в одной микросхеме) и работает самостоятельно в нашем ЗУ.
2. Компилятор имеет очень много настроек, которые передаются через строку параметров или в тексте программы, поэтому его удобно использовать через оболочку, которая красиво отображает программу и позволяет рулить настройками компилятора, а также может показывать результаты работы компилятора, использоваться в качестве отладчика программы, может прошивать процессор, если в комплекте с компилятором идет соответствующая утилита.
3. Компиляторы бывают от разных производителей и бывают разные сборки у одного производителя. Все они работают по разному и имеют разные глюки.
4. За компилятор как и за любую программу ни один автор не дает гарантий, поэтому, учитывая все многообразие возникающих проблем при программировании процессоров для различных устройств, вся ответственность лежит на нас и мы ее никогда не сможем оправдать на 100%, как бы мы не старались, поэтому наше программирование должно быть максимально надежным и предусмотрительным.
5. После компиляции программы, всегда надо смотреть на результаты работы компилятора. На сколько удачно ему удалось выполнить свою работу и разместить в памяти программу, переменные, стек, постоянные данные. Нужно создать запас памяти и предусмотреть наиболее простую структуру программы и в тоже время все еще удобную для чтения и написания.

Как же работает компилятор? Это приблизительное описание:

1. Берется главный файл проекта и выполняются все команды препроцессора, т.е. внутрь главного файла набивается вся программа целиком со всеми файлами утилит и всеми файлами описателей и получается огромный файл единой программы.
2. Выполняется первый проход по файлу выявляя все ошибки - лексический анализ, правильность расстановки разделяющих знаков, проверка скобок и т.д. проверяются простейшие правила, которые можно проверить не вникая в глубинный смысл программы.
3. Выполняется синтаксический анализ. Вся программа превращается в древовидную структуру (как реестр в Windows) или в нечто подобное с целью вынуть смысл программы из текста, разбить все по понятиям, кто кому приходится детьми и родителями и проверить соответствие типов, количества параметров, установить взаимосвязи между описанием и вызовом. Древовидный тип считается самым удобным для описания сложных логических построений.
4. Оптимизация. Все это дерево анализируется на предмет лишних, подвешенных в воздухе веток и они отбрасываются или не отбрасываются (не всегда компилятор может понять все взаимосвязи) Происходит упрощение и обобщение кода.
5. Из этого голого смысла формируется текст программы в машинных кодах.

Сначала мелкие подпрограммы превращаются в код известной длины. Потом все маленькие подпрограммы расставляются в памяти линковщиком, который вставляет перекрестные ссылки на подпрограммы. На этой стадии тоже происходит оптимизация по размеру или скорости. Возможно некоторые куски программы жмутся архиватором и где-то в тексте добавляется деархиватор.

Вспомним что в ATMega32 три вида памяти

FLASH (здесь лежит программа и константы: таблицы символов, схема меню, все текстовые фразы),

ОЗУ (здесь переменные, массивы, стек),

ПЗУ (настройки пользователя, настройки аккумуляторов и химии).

Компилятор делает файл с программой на машинных кодах, который выглядит как последовательность байтов, которая будет записана во FLASH. Компилятор также может сделать файл для ПЗУ с начальными установками аккумуляторов и химии, но компилятор ничего не делает для ОЗУ. ОЗУ при выключении полностью теряет все и при включении там неизвестно что - мусор.

Процессор выполняет программу, которая лежит во FLASH и никогда не меняется, а вот в ОЗУ творится жизнь, там непрерывно все меняется: меняется содержимое глобальных переменных, возникают и исчезают временные переменные, а в конце памяти растет и убывает стек. Верх и низ ОЗУ никогда не должны пересечься. А в ПЗУ что-то меняется только, если пользователь сделал изменение параметров.

Компилятор очень точно просчитывает где в ОЗУ и что будет храниться. Сначала идет область переменных и массивов глобальных (видных из всех функций и подпрограмм и живут всегда), потом идет область временных переменных (живут только на время выполнения функции), которая растет в сторону конца, потом идет пустота и весь конец ОЗУ используется под СТЕК.

СТЕК начинает расти от конца к началу. То растет, то убывает. Это зависит от глубины ныряния процессора внутрь подпрограмм и от количества передаваемых параметров через функции, но самая глубина и самый большой размер стека, когда в самой глубине подпрограммы вызывается прерывание, которое спасает все состояние процессора в стек и занимает пространство ОЗУ под временные переменные. А ОЗУ у нашего процессора всего 2048 байт и это ну прям впритык мало, и мало 32768 байт под программу FLASH. Вот почему я такой жадный, потому что глупый, и не могу вместить удобство и функциональность, которые мне так важны.

После работы компилятора winavr avr-gcc (GCC) 4.2.2 (WinAVR 20071221rc1) выдается сообщение:

```
Program: 30316 bytes (92.5% Full) (.text + .data + .bootloader)
Data: 1041 bytes (50.8% Full) (.data + .bss + .noinit)
EEPROM: 914 bytes (89.3% Full) (.eeprom)
```

Это означает что программа работать будет, потому что места хватит.

А вот после работы последней версии компилятора и 1 и 2 строка зашкаливают за 100%, такую программу даже не пытайтесь прошивать. Возможно я что-то упустил с настройками компилятора. По идее последний компилятор должен быть лучше раннего.

Одновременно с созданием машинного кода компилятор создает файл main.lst и main.lss, в которых вы можете найти программу на ассемблере и таблицы размещения функций, переменных и констант.

Мы приблизительно знаем как работает "инструмент" - язык Си. Теперь надо включить фантазию и увидеть какой будет наша программа для ЗУ (максимально лучшее, что мы можем пожелать на сегодня).

## Урок №13 (Рецепт как написать программу для ЗУ)

Наша программа для ЗУ должна уметь следующее (в порядке мысленного взгляда на чужие ЗУ):

1. Понятное и безопасное меню без всяких там инструкций (на сколько позволит размер программы).
2. Быстрый и удобный запуск зарядки.
3. Исчерпывающая информация о результатах, которая никуда не пропадет при нажатии не на ту кнопку.
4. Доступ к настройкам.
5. Простота.
6. Крутой и гибкий алгоритм зарядки. Проверка на неестественное поведение аккумулятора.
7. Связь с большим компьютером и возможность отдать ему вообще всю информацию из ЗУ онлайн.
8. Возможность без большого компьютера проводить настройку и отображение ошибок и всяческих переменных.

Не смотря на то, что главный смысл ЗУ - качественно заряжать, большая часть программы будет посвящена интерфейсу с пользователем. Также и в автомашинах: главное это двигатель, но дороже всего стоит железный корпус и удобный салон.

Сам алгоритм зарядки - это проверка текущего состояния датчиков, вычисление необходимой коррекции и посылка команд на исполняющее устройство (установка текущего тока зарядки) займет очень мало места. Если бы мы располагали исследованиями в области аккумуляторов да на русском языке, да имели экспериментальную базу, можно было бы нагородить огород с использованием скоростей температуры, напряжения или производных dV/dI dT/dV и т.д. И все равно существенно это не заняло бы места, если только там нет волшебных таблиц с критическими точками.

Имея дело с широким списком различных производителей и будучи ограниченными в памяти процессора, путь развития алгоритма зарядки остается возможным, но пока не приоритетным.

А вот имея "широкие массы" пользователей не гоняющиеся за максимальной отдачей аккумулятора и часто делающие ошибки, мы имеем приоритетное направление удобный интерфейс. Кроме того, удобный интерфейс поможет отлаживать нашу программу без отладчика и настраивать ЗУ.

Вывод: Первое чем займемся - это интерфейс на все случаи жизни с минимальной загрузкой процессора и с минимальными, на сколько это возможно, затратами на его использование.

Должна получиться заготовка, которую можно будет легко использовать для любого устройства на At Mega с прицепленным ЖКИ и с очень простыми функциями отображения информации на ЖКИ. Для использования нашей заготовки на любом самодельном устройстве, надо, всего лишь, переопределить ножки процессора к которым подключено ЖКИ и писать программу с отображением отладочной информации на экран для самоконтроля.

Как при создании программ для ЗУ, так и при создании любых программ, можно заметить, что на экране появляются одни и те же изображения, которые легко обобщаются. А так как мы экономим место и время, то это нам поможет.

Лирическое отступление как делаются архиваторы: Когда то давно мне понадобилось сократить трафик между моими двумя программами, которые были удалены

друг от друга. Читая про способы архивации, я наткнулся на заумные рассуждения.

И, как обычно, долго разбираясь в этом талмуде, оказалось, что все можно объяснить на пальцах. Во всех методах сжатия информации используется один и тот же принцип: очень умные дяди внимательно смотрят на эту информацию и ищут закономерность. Если закономерность есть, значит данные сжать можно, если закономерности нет, нет. Если какие то фрагменты данных повторяются, значит их можно один раз запомнить, а в то место где они стоят (в архивированном файле) поставить ссылку на имеющуюся одну копию (ссылка должна быть меньше по размеру чем повторяющийся фрагмент).

Именно этот способ мы применим для функций - все подобные куски программы мы выделим в одну функцию и будем ее вызывать.

Тоже самое для меню - мы сделаем шаблоны экранов. Один раз опишем их. А рисованием шаблонов будет заниматься одна функция, потому что все шаблоны будут построены в соответствии с единым "правилом".

Как архиваторы смотрят на данные, точно также программист смотрит на свою программу и ищет, где он повторился и, нельзя ли это место обобщить и один раз описать и потом многократно использовать.

Точно также каждый человек должен подумать про устройство мира. Если есть одинаковые события их надо обобщить и придумать правило/закономерность, чтобы работать не с самими событиями, а с их закономерностью. Не надо запоминать устройство каждого магазина, а надо знать их общее. Надо найти закономерности жизни - правила жизни - правила успешной жизни. Надо найти цель жизни. Надо использовать правила для достижения цели или того места где она приблизительно находится, а там сориентироваться. Надо отмести все лишнее и вредное, надо перестать врать себе и отворачиваться от ясных рассуждений. Надо следовать ясным, добрым рассуждениям, иначе нормальной программы не написать.

Вывод информации на ЖКИ будет как бы состоять из 3 уровней:

1. Самый высокий уровень (программиста) в тексте программы происходит формулирование требований к шаблону (передача параметров) и вызов самого шаблона по его номеру. Все шаблоны нами будут перенумерованы и заполнены заранее. Шаблон может обновляться сам, отслеживая изменения переменных.
2. Уровень функции заполнения шаблона данными. При вызове шаблона с параметрами функция отрисовки шаблона заполняет шаблон текущими данными (фразами из словаря и переменными I,V,T...). Прорисованный шаблон помещается в видеопамять в ОЗУ в виде букв и символов для всех строчек ЖКИ. А также сигнализирует, что произошло обновление видеопамяти.
3. Самый низкий уровень. Уровень прерывания производит передачу видеопамяти в ЖКИ со скоростью не более 1 экран целиком за 0.5 секунды. Причем прерывание передает за 1 сеанс не более 1 команды (ЖКИ) или буквы (ЖКИ) так, чтобы задержка отработки ЖКИ попадала на время работы основной программы.

Все ожидания реакций ЖКИ наш процессор не ждет, а работает с основной программой. Вот она хитрость экономии. ЖКИ после приема буквы или команды задумывается, это время ему нужно для записи буквы в свою память. Так вот наш процессор должен ждать (пока там ЖКИ скушает букву), чтобы передать следующую. А мы ждать не будем. Передадим букву и закончим прерывание. Пока ЖКИ жует, мы выполняем основную программу.

Т.о. нам нужно вызывать "видео-прерывание" 4строки * 20символов * 2раза_всекунду * 2раза_прозапас=320 раз в секунду.

Т.о. как во взрослых компьютерах у нас есть видеопамять и видео прерывание (драйвер видеокарты), только видеопамять мы используем из ОЗУ и самой прорисовкой

видеопамяти занимается сам процессор, а не посторонняя видеокарта. Но, в целом, все идеи те же. Мы выполняем нашу второстепенную задачу: глобальная экономия всего. За счет усложнения структуры программы, мы сэкономили время (ожиданий) и память программы (на многократных вызовах функции посылки фраз в ЖКИ).

Чуть по подробнее расскажу про шаблон. Например, при зарядке аккумулятора на экране отображается информация о ходе процесса зарядки. Такая информация постоянно отображается и обновляется в течение нескольких часов как для 1 канала так и для 2 канала. Циферки меняются а форма шаблона одна и та же, значит форму шаблона можно жестко описать. Те же рассуждения относительно всей отображаемой информации на ЖКИ. например экраны настройки или экраны отражения итогов зарядки или само главное меню, несмотря на то что разное, имеет в себе нечто общее.

Чтобы описать шаблон мы придумаем коды-команды шаблона например такие:

```
// Шаблоны экрана
// Описываются байтами - командами
// 0xFF - Конец шаблона
// 0      fc - Поставить курсор
//        fc=0xff -Курсора нет нигде CURSOR_OFF
//        7b=1    -Курсор моргает квадратом CURSOR_BLINK_ON
//        7b=0    -Курсор горит подчерком CURSOR_LINE_ON
//        10b     -Номер строки где стоит курсор
//        65432b  -Номер столбца где стоит курсор
// 100    fc=0xff -Курсора нет нигде CURSOR_OFF

// 1                - Очистить весь экран
// 2                - Очистить первую строку
// 3                - Очистить вторую строку
// 4                - Очистить третью строку
// 5                - Очистить четвертую строку
// 6     XY NN      - Очистить строку начиная с XY длинной NN
// 7     XY NN KK   - Напечатать переменную KK начиная с XY целым десятичным
числом длинной NN
// 8     XY NN KK   - Напечатать переменную KK начиная с XY целым
шестнадцатиричным длинной NN
// 9     XY MN KK   - Напечатать переменную KK начиная с XY FLOAT числом длинной
M включая N знаков после запятой
// 10    XY NN KK   - Напечатать бинарную-переменную KK начиная с XY
// 11    XY KK      - Напечатать время-переменную KK начиная с XY
// 12    XY KK NT   - Взять значение из переменной, KK добавить к нему NT это
будет номер текста для печати
// 20    XY KK      - Напечатать название аккума
// 21    XY KK      - Напечатать название типа аккума
// 14    XY na tt   - Напечатать байт na из временного массива акк
// 15    XY nt tt   - Напечатать байт nt из временного массива типов акк

// 16    XY TT      - Напечатать текст TT начиная с XY
// 22    XY f       - Напечатать название акк(f=1) или типа акк(f=0) из
временного массива
// 17    XY MM      - Напечатать меню MM начиная с XY
// 18               - Изображаем 3-х уровневое меню
// 19               - Изображаем 1-но уровневое меню

// tt =
//       tt=0 двоично
//       tt=1 десятично
//       tt=2 шестнадцатирично
//       tt=3 флоат
//       tt=4 флоат/50
//       tt=5 флоат/10
// XY=X(5 бит старших)+Y(3 бита младших)
```

Чем круче список команд шаблона, тем больше возможностей понравиться

пользователю. Как видите, идет тотальная экономия битов и байтов, поэтому X смешивается с Y для отражения адреса знакоместа на ЖКИ.

Выбор принципа шаблонов - это достаточно сложный метод. Если ваша программа из 5 разных шаблонов, то, возможно, от применения шаблонов никакой экономии не произойдет, но если у вас шаблонов много...

С вызовом рисования шаблона все ясно - ЖКИ все время что то да рисует, чтобы пользователю не было скучно.

Заведем глобальную переменную BYTE, которая содержит в себе код текущего шаблона. Если вы покопаетесь в программе, то увидите, что для того, чтобы вызвать рисование шаблона, я использовал функцию **SH()**, заведенную для препроцессора .

Например мне надо, чтобы в каком то месте программы начал выводиться шаблон номер 5, тогда я в тексте программы пишу **SH(5)** и, даже, без обычной для Си ";" в конце оператора. Препроцессор перед компиляцией изменяет мою инструкцию на следующее выражение **{nSh=5; pSh=255; cSh=2; sSh.first=1;}** Это делает директива препроцессора **#define SH(n) {nSh=n; pSh=255; cSh=2; sSh.first=1;}**

**nSh=5;** - это тот самый код текущего шаблона номер 5.

**pSh=255;** - этот шаблон нарисовать 1 раз, если бы <255, то перерисовывать шаблон каждые **pSh/10** секунд.

**cSh=2;** - счетчик для отсчитывания **pSh** со временем уменьшается, ему специально присваивается ненулевое значение, т.к. прерывание могло его уже уменьшить.

**sSh.first=1;** - перерисовать шаблон полностью как в первый раз.

Для управления шаблонным механизмом и вообще всем ЖКИ из основной программы я использую всего несколько команд, указанных ниже, все остальное делает за меня функция рисования шаблонов в видеопамять и прерывание, отправляющее видеопамять в ЖКИ. Не пытайтесь вникнуть в Си этих команд, просто почитайте комментарий и обратите внимание на первое слово после #define

```
// Пометь строки в которых были изменения
#define XYL switch((XY)&7){case 0: fv.s1=1; break; case 1: fv.s2=1; break; case
2: fv.s3=1; break; case 3: fv.s4=1; break;}
// Сделать текищим шаблон n период перерисовки изменяемой части p
#define SHABLON(n, p) {nSh=n; pSh=p; cSh=2; sSh.first=1;}
// Сделать текищим шаблон n период перерисовки изменяемой части p
#define SH(n) {nSh=n; pSh=255; cSh=2; sSh.first=1;}
// Отобразить шаблон меню
#define MENU nSh=1; pSh=255; cSh=2; sSh.first=1;
// Сейчас перерисовать текущий шаблон полностью
#define REFRESH_SHABLON cSh=2; sSh.first=1;
// Сейчас перерисовать только изменяемую часть шаблона
#define REFR_SHABLON cSh=1;
// Поставить курсор и заставить моргать или неморгать
#define CURSOR(x, y, type) (type==CUR_OFF)?fc=255:fc=((y)&3)+(((x)&31)<<(2))+
(type==BLINK_ON?128:0);
// Установка курсора при редактировании названия аккума или типа аккума для 4х
строчного ЖКИ
#define FC(x) ((x)<<(2))+3+128;
// Рассчитать XY из x и y
#define XYN(x, y) ((x)<<(3))+((y)&7)
```

А вот как выглядит настоящий шаблон номер 3 (одна строка одна команда шаблона). Берем первую циферку из скобок и смотрим что она означает в командах чуть выше описанных.

```
// Шаблон отражения зарядки/разрядки первого канала (название аккума ток напряжение температура заряд в
процентах время)
FCHAR _s3[]={1,                         // Очистить весь экран
             12, XYN(0, 0), 108,  16,  // Рисовать текст 16+(108 перем.) «зарядка/разрядка» 0столбец 0строка
             16, XYN(0, 1), 52,        // Рисовать 52 текст "I1(A)=" 0столбец 1строка
             9,  XYN(6, 1), 0x52, 101,// Печатать float на 5 позициях 2 знака после запятой 6столбец 1строка
             16, XYN(12,1), 92,        // Рисовать 92 текст "E%=" 12столбец 1строка
             9,  XYN(15,1), 0x51, 109,// Печатать float на 5 позициях 1 знак после запятой 15столбец 1строка
             16, XYN(0, 2), 46,        // Рисовать 46 текст "V1(B)=" 0столбец 2строка
             9,  XYN(6, 2), 0x52, 102,// Печатать float на 5 позициях 2 знака после запятой 6столбец 2строка
             11, XYN(12,2), 104,       // Напечатать время 1го канала 12столбец 2строка
             16, XYN(0, 3), 90,        // Рисовать 90 текст "T1(oC)="0столбец 3строка
             9,  XYN(7, 3), 0x52, 103,// Печатать float на 5 позициях 2 знака после запятой 7столбец 3строка
             100,                      // Отключить курсор, чтоб нигде не моргал
             255};                     // Конец шаблона
```

Как видите, 80 символов абсолютно разной ценной информации многократно отражаются на экране ЖКИ. На все потрачено 37 байт, подпрограмма рисования шаблонов и прерывание на вывод в ЖКИ. С учетом большого многообразия шаблонов, это достаточно экономное использование памяти.

Хотя вся эта информация глубоко специфичная и почти никому не пригодится, но даже те, кто никогда программировать не будут, могут понять общий подход и использовать его где-то в других областях своей деятельности.

## Урок №14 (Работа с памятью)

Сначала небольшое обобщение относительно больших, нормальных компьютеров и программирования на Си. Начну из далека.

Компьютером называется совокупность трех компонентов: процессор, память, периферия, доступная через порты ввода вывода. До нынешних времен под памятью в основном имели ввиду ОЗУ (Оперативное Запоминающее Устройство, SIMM, DIMM, SDRAM, DDR). Эта та самая память, которая теряет все при выключении, эта та самая память в которой ячейка памяти - мизерный конденсатор, который теряет свой заряд за доли секунды если его не подпитывать (регенерация памяти). Зато такая память обладает максимальным быстродействием и одна ячейка занимает на кристалле минимальное место. Плата за такие супер параметры - регенерация. На материнской плате есть специальный контроллер памяти, который периодически читает всю память и записывает (подпитывает) это же значение в память. Это делается параллельно обычной работе компьютера, поэтому вообще не заметно для пользователя.

Зачем же нужна такая память, которая все забывает? Зачем нужен народ который не помнит своей истории, не помнит своих предков? Не помнит уроков: как делать нельзя и как можно. Плохая аналогия, но что то есть.

Оказывается, что такая память вполне подходит компьютеру для хранения ПЕРЕМЕННЫХ. Ни одна программа больше трех пальцев не может существовать без переменных. Чтобы посчитать а+б=с, необходимо где-то временно хранить а и б. Потом они уже не нужны, нужен результат. Результат будет высечен в мраморе на века, а временное забудется и уйдет. Но без этого временного компьютер работать не может. Любой процессор имеет кусочек ОЗУ прямо внутри себя. Такая внутренняя память называется регистры процессора - это самая быстрая память, самая необходимая, самая главная. Все логические операции совершаются там в голове, а потом хранятся в переменных в ОЗУ в более медленной памяти чем регистры, потому что надо оттуда читать, а на это уходит время.

Нынешние процессоры имеют дополнительно к регистрам CASH - особую память в которую закачивают копию фрагмента ОЗУ наиболее часто используемую, чтобы сэкономить время, но это все детали.

Язык Си работает в основном с обычной ОЗУ. Там создаются временные переменные необходимые для работы программы. Обращение к памяти происходит через обращение к переменным и массивам, размещенным в памяти. К памяти также можно обращаться прямо по адресу, но это надо делать осторожно (можно порушить данные системы), поэтому не приветствуется.

Процессор работает не только с ОЗУ, но и с другими устройствами: с клавиатурой, мышкой, видео картой, сетевой картой, COM-устройством, USB-устройством, винчестером, CD-ROM, DVD, BDROM, принтером, плоттером, ЗУ, дисководом, флэшкой, мобилой, фотиком и т.д. Все это периферия. Большинство этих устройств имеет единый механизм обращения к ним.

Работа всех вышеперечисленных устройств зависит от параметров их работы, а это информация в виде байтов. Например: раскрутите диск до скорости 2000 об/мин, поставьте головку 1 на 145 цилиндр, прочитайте 49 сектор, информацию из сектора дайте мне. Все команды и вся прочая информация побайтно передается через порты ввода/вывода. И тот самый универсальный механизм - это механизм ПОТОКА ДАННЫХ. С памятью ясно как работать либо переменные либо прямая адресация, а со всеми неизвестными устройствами, через их драйвера открывается поток данных.

"Все" драйвера "всех" устройств обязаны поддерживать стандартный набор команд, чтобы Windows мог дать доступ через системные процедуры к ним. А Си как раз прицепляется к системе и знает как с ней разговаривать. Си общается только с системой. Система общается через дрова со всей периферией.

Так вот Си дает доступ к любому устройству через стандартный механизм потока с ограниченным набором команд. Вся ненужная информация и детали прячутся от юзера остается самое необходимое передача и получение данных из переферии. Где-то в этих данных может быть и управляющая информация.

1. Открыть поток и получить указатель на него. Далее во всех командах обязательно есть указатель на поток.
2. Получить длину потока в байтах если есть.
3. Встать на определенный элемент потока.
4. Прочитать или записать данные в поток с текущего элемента.
5. Закрыть поток.

Очень интересно шло развитие компьютерной техники - от малого ко все более большому, от частного к общему и потом, имея опыт сложного возвращаемся к частному маленькому и видим как в малой неполной структуре отражены глобальные законы всего мира.

Это мы говорили про большие компьютеры, чтобы иметь общее представление, теперь вернемся опять к ATMega32.

У ATMega32 три вида памяти ОЗУ(RAM), ППЗУ(EEPROM)(перепрограммируемая постоянное запоминающее устройство), ФЛЭШ(FLASH). Во ФЛЭШ лежит программа и некоторые данные для рисования на ЖКИ, в ОЗУ лежат переменные, в ППЗУ лежат настройки пользователя для аккумуляторов.

WinAvr Си работает с ОЗУ так же как и в больших компьютерах просто и легко, а вот FLASH и EEPROM имеют особенности. Во первых, 3 вида памяти и все имеют свою адресацию, у каждой памяти есть свой 0 и свой конец. Во вторых, команды чтения и записи разные, поэтому компилятор должен знать кто есть кто чтобы знать как писать туда.

В компиляторе IAR Си все так и сделали. Описываем переменные, указывая где они лежат, и все, дальше просто работаем с переменными, не думая ни о чем. В WinAVR к сожалению это не так. Не только надо описать переменные, но и работать с ними особым

образом, своими функциями записи и чтения для EEPROM(читаем и пишем) и FLASH(только читаем). С ОЗУ все как обычно проблем нет, все как в стандартах Си для любых компьютеров.

Ниже я приведу свои функции работы с EEPROM и FLASH. Начнем с FLASH:

Во FLASH у нас хранится:

1. Все фразы меню и массив указателей на начало каждой строчки меню
2. Все фразы шаблонов и массив указателей на начало каждой фразы
3. Ноты музыки
4. Таблица перекодировки ASCII символов в кодировку ЖКИ

Со всей этой информацией мы работаем через нижеуказанный механизм чтения из FLASH

```
// Подключаем стандартный комплект функций WinAVR для работы с FLASH
#include <avr/pgmspace.h>           // Работа с FLASH

// Описание байта во флэш памяти и то и другое по сути одно и тоже
// просто в разных подпрограммах из WinAVR требуется строго один из двух
// приходится подстраиваться под бесплатные дары
// Опять же назвал по своему, т.к. ихнее название некрасивое не логичное
typedef prog_uchar FBYTE;           // Байт во FLASH памяти
typedef prog_char FCHAR;            // Символ во FLASH памяти

// Из флэш памяти только читаем.
// Вот все функции чтения флэш
// Я их немного переобозвал, чтоб все логично и компактно было
#define FRB  pgm_read_byte          // Читаем байт из FLASH
#define FRW  pgm_read_word          // Читаем ссылку из FLASH
#define FRS  memcpy_P               // Читаем кусок из FLASH

// Например так я описал меню
// Сначала описываем указатели на фразы и размещаем сами фразы в памяти
FCHAR _m0[]="I.Канал 1";
FCHAR _m1[]="II.Канал 2";
FCHAR _m2[]="III.Аккумуляторы";
// Потом собираем указатели на фразы в массив указателей
PGM_P PROGMEM Mn[3]={_m0, _m1, _m2};

// Теперь, зная какую фразу нам надо вывести (ее номер), мы по номеру достаем
// указатель, а по указателю(адресу первой буквы фразы) копируем буквы
// например в видеопамять которая будет прерыванием выведена в ЖКИ.
PGM_P fadr=(PGM_P)FRW(&Mn[1]);   // Читаем указатель(адрес) первой фразы
BYTE x=FRB(fadr);                // Можем прочитать первую букву из этой фразы
FRS((BYTE*)V2, (PGM_P)fadr, len);// Копируем len байтов фразы в видеопамять
```

Как я ранее писал, есть 3 уровня "низости":

1. Верхний уровень - работа с шаблонами (очень простой способ полностью разрисовать ЖКИ пятым шаблоном: SH(5))
2. Второй уровень - прорисовка шаблонов в видеопамять. Как раз этот уровень сегодня обсуждаем.
3. Низкий уровень - уровень прерывания. Посылка видеопамяти в ЖКИ.

На нашем нынешнем уровне развития считаем что: Ссылка=Указатель=Адрес

Во FLASH у нас хранится:

1. Все фразы меню и массив указателей на начало каждой строчки меню
2. Все фразы шаблонов, массив указателей на начало каждой фразы

3. Все шаблоны и массив указателей на начало каждого шаблона
4. Ноты музыки
5. Таблица перекодировки ASCII символов в кодировку ЖКИ

У нас 3 массива указателей: на начало каждой фразы меню, на начало каждой фразы шаблона и на начало каждого шаблона. Эти массивы указателей нужны нам, чтобы, по номеру, найти в памяти начало каждой фразы или шаблона.

Например SH(5) - рисуем пятый шаблон. Программа берет 5тый указатель на шаблон из массива указателей, а указатель это адрес в памяти, значит программа знает откуда из памяти брать байты 5-го шаблона. Программа берет по очереди байты шаблона, пока не наткнется на 255(конец шаблона). И вот встречена команда 16 (напечатать фразу шаблона 95). Опять программа достает 95-тый указатель из массива указателей на фразы шаблона, а этот указатель это адрес первой буквы фразы. И тогда программа берет по очереди все буквы пока не встретится 0 (признак того что фраза закончилась) и копирует все эти буквы, подменяя их соответствующим кодом для русской буквы ЖКИ в видеопамять.

Вот таким хитрым образом мы объяснили работу с FLASH, рассказали, что мы там храним и немного объяснили второй уровень "низости".

Теперь EEPROM: Зачем нужно было городить огород с тремя видами памяти, знает только производитель ATMega32. Вероятно они руководствовались соображениями экономии денег. По большому счету EEPROM и FLASH это одно и тоже. FLASH вероятно подешевле будет, но количество циклов перезаписи меньше и возможно занимаемое место на кристалле меньше. EEPROM более тормозная, больше циклов перезаписи. Для нас для программистов и то и другое это постоянная память и лучше бы она была одна, а не две. Но деваться не куда придется работать с тем чем есть.

В EEPROM мы храним:

1. Настройки химии аккумуляторов. Каждая химия ведет себя одинаково.
2. Настройки аккумуляторов. Сколько банок и ампер-часов в моих любимых аккумуляторах.
3. Настройки интерфейса ЗУ. (Музыка)
4. Настройки каналов (подстроечные коэффициенты для расчета I,V,T из тех вольт, которые намерены АЦП)

Все эти данные можно читать и писать, но желательно не часто. И скорость чтения и записи будет не быстрая. Поэтому при включении ЗУ или запуске Канала мы все считываем в ОЗУ и с этими данными работаем. Иногда при настройке ЗУ запишем подстроечные коэффициенты. Изредка добавим новый аккумулятор. Короче процедуры записи и чтения EEPROM вызываются редко и медленно.

```
// Процедуры работы с EEPROM от WinAVR стандартная поставка
#include <avr/eeprom.h>              // Работа ЕЕПРОМ

// Описание размещения данных (Химия Акки Коэффициенты)
// На основании его компилятор создает прошивку main.eep для EEPROM
#include "eeprom.h"                  // Описание содержимого ЕЕПРОМ

// Функции для записи и чтения EEPROM переназванные по человечески
#define ERB  eeprom_read_byte        // Читаем байт из EEPROM
#define ERS  eeprom_read_block       // Читаем кусок из EEPROM
#define EWB  eeprom_write_byte       // Пишем байт в EEPROM
#define EWS  eeprom_write_block      // Пишем кусок в EEPROM

EEMEM BYTE ET1=1; // eeprom.h (0-датчик отсутствует, 1-датчик работает)
a=ERB(&ET1);      // Прочитали из EEPROM в обычную переменную BYTE ОЗУ
EWB(&ET1, a);     // Записали обычную переменную BYTE в EEPROM

// float это 4 байта, поэтому читаем его из EEPROM и пишем в ОЗУ переменную
// т.е. копируем 4 байта начиная с адреса в EEPROM в ОЗУ туда где расположена
// нужная нам переменная
EEMEM float E1_k0=-0.302066773176193; // описано в eeprom.h

ERS((BYTE*)&Ch1.k0, (BYTE*)&E1_k0, 4); // Читаем из EEPROM коэфф тока 1 канала
EWS((BYTE*)&Ch1.k0, (BYTE*)&E1_k0, 4); // Пишем в EEPROM коэфф  тока 1 канала
```

Для новеньких в Си:

Многие функции, как в вышеописанных примерах, для копирования из одного вида памяти в другой вид памяти требуют чтобы им указали адрес откуда куда. Поэтому в Си придуман специальный значок "&" который ставится перед переменной спереди, а компилятор когда делает программу на машинных кодах на место "&переменная" вставляет адрес этой переменной. Слово (BYTE*) подсказывает компилятору, что это будет именно адрес последовательности байтов.

В нижнем примере мы обязаны добавить это слово потому что сама переменная была описана float, а функция ждет BYTE. Компилятор сразу предполагает что мы сделали ошибку и подсунули ему float вместо BYTE, а мы ему явно говорим: Будь спокоен это то что нужно, потому что float это 4 штуки BYTE.

## Урок №15 (прерывания погружение)

Не смотря на то, что на дворе 21 век, до сих пор встречаются программы написанные для DOS и такие программы исходят например из ЦБ РФ (SPRAV.MFO). Вероятно еще не вымерли программисты старой закалки. В таких программах опрос клавиатуры производится на частоте процессора. Представьте себе, что процессор 3 миллиарда раз в секунду опрашивает клавиатуру "не нажата ли какая кнопка". Это при том, что пользователь за весь день лазанья по интернету, едва ли, 100 кнопок нажал. Даже самые продвинутые секретари набирают не более 100 кБ в день, нажимая кнопки в среднем не чаще 1 в секунду. Но если, вдруг, пользователь заметил, что отклик компьютера на нажатие задерживается на 1/3 секунды, то говорят что клавиатура тупит. Причем здесь клавиатура?

На одно ядерных компьютерах такая программа DOS парализует работу всех остальных программ. Для того чтобы в программах не делать множество циклов, ожидающих происхождения какого либо события (нажатия кнопки, прихода широковещательного запроса по сетке, звонка по телефону, наступления 1000-ной секунды, движения мышки и т. д.), мы можем использовать прерывания. Прерывания встроены в логику процессора аппаратно. Прерывания прерывают основную программу телевидения для важного сообщения, т.е. запускают программу обработки прерывания. Прерывания

наступают внезапно или циклически. Прерывания это появление сигналов на определенных ногах процессора или особые "прозрения" процессора.

В моменты прозрения в голове процессора открывается новое видение мира, открывается истина, которая настолько фундаментальна, что сиюминутные заботы процессора кажутся ничтожными. Осознание истины очень приятно. Что происходит в этот момент не понятно. Происходит какое то созерцание/восприятие истины, мысли останавливаются, всякие там рассуждения о выгоде и прибыли тем более.

Истина приходит в виде цельного образа, который испаряется и в остатках тумана угадывается ответ на давно мучивший вопрос. В следующее мгновение уже ничего нет. И доказательств нет, но ответ как ни странно правильный.

В ATMega32 есть такие прерывания:

1. Прерывание при сбросе или включении.
2. Внешнее прерывание 0,1,2. (Программируется работа 3х ног процессора специальным образом): при возникновения на ногах процессора +5в или 0в срабатывает прерывание.
3. Счетчик/таймер 0,1,2. Внутри процессора есть счетчики, которые считают кратно тактовой частоте, при достижении счета определенного числа или при достижении максимального значения (переполнения) срабатывает прерывание.
4. При работе с последовательной передачей данных (например от ЗУ к большому компьютеру или обратно) возникает необходимость подготовить новые данные для отправки или спрятать в память полученные данные, для этого используются прерывания "пришел последний бит очередного байта", "отправлен последний бит отправляемого байта", "последовательная передача завершена", "байт данных при передаче пуст".
5. Завершена трансформация Аналогового сигнала в цифровой.
6. При записи или чтении в ЕЕПРОМ "ЕЕПРОМ готов".
7. Сравнение аналогового сигнала и моего сторожа(цифрового) завершено.
8. Передача последовательных данных по 2 проводам ОК.
9. Запись во ФЛЭШ завершена. Как мы уже говорили в первых 40 байтах ФЛЭШ (памяти программы) лежат "вектора прерываний" (адреса подпрограмм обработки). При срабатывании прерывания, выполняется соответствующая подпрограмма.

Нам конечно же понадобится прерывание "АЦП завершено" и прерывания по передаче и получению байтов в COM-порт. Но, кроме того, у нас есть прерывание циклическое (по переполнению счетчика) для передачи низких команд в ЖКИ. Циклическое прерывание надо нам, чтобы играть музыку и отсчитывать секунды зарядки.

Вот магические слова которые описывают прерывания в программе, чтобы компилятор правильно вписал вектора куда следует.

```
// Описатели прерываний
#include <avr/interrupt.h>          // Работа с прерываниями

// Прерывание переполнения счетчика 0
ISR(TIMER0_OVF_vect)
{
}

// Прерывание "Преобразование АЦП завершено"
ISR(ADC_vect)
{
}

// Это прерывание обрабатывает поступившие команды с компьютера по COM порту
ISR(USART_RXC_vect)
{
}

// Отправка очередного байта на COM-порт большого компа
ISR(USART_TXC_vect)
{
if(iOutBuf)                     // Если обратный счетчик не пуст
  {
  UDR=OutBuf[nOutBuf-iOutBuf]; // Посылаем следующий байт
  iOutBuf--;                    // Уменьшаем счетчик буфера
  }
else                            // Иначе все уже отправили
  fTran=false;                  // С посылкой последнего байта зак транзакция
}
// Сюда будут направлены все неиспользованные прерывания (может быть)
ISR(BADISR_vect){}
```

Все тексты вы можете посмотреть в исходниках. Привожу только текст отправляющего на большой компьютер прерывания, т.к. он прост и показывает, что в прерывании надо действовать быстро.

## Урок №16 (ЖКИ)

ЖКИ - это доисторическое графическое, текстовое (знакогенерирующее) устройство, оказавшееся очень живучим. Уже давно придуманы экраны мобильников, которые могут несравненно больше, стоят дешевле и вообще... И все же ЖКИ до сих пор пользуются популярностью у разработчиков всего мира из-за своей простоты. Можно было бы доработать эту простоту до еще большей простоты, но этого никто не делает. Лучше иметь что-то постоянное, хотя и немного несовершенное.

ЖКИ - это стандарт HD44780.

ЖКИ - это автономный компьютер, в котором есть своя память, свой процессор, свои порты ввода вывода и экран. У ЖКИ есть своя ОЗУ - видеопамять, которая непрерывно аппаратно отражается на экране.

ЖКИ работает на частоте 1 МГц.

ЖКИ имеет параллельную, 8-ми и 4-х битную шину передачи данных.

ЖКИ - это устройство для отображения текстовой информации и соответственно его функции заключаются в том, чтобы получить эту информацию из шины и отобразить ее на экране. Больше ничего ЖКИ не может. Если бы мы были создателями стандарта общения с ЖКИ - HD44780, то как бы мы его написали?

Вероятно, должна быть команда полный перезапуск и чистка всего. Должны быть команды нарисовать букву там где стоит курсор и передвинуть курсор на следующую позицию, переставить курсор вообще на произвольную позицию. Должны быть команды, управляющие типами курсора и команды работы с псевдографикой. Так оно и есть.

Если мы знаем функции (смысл) устройства, то и как им управлять в общих чертах нам тоже ясно, потому что мы такие же разработчики как и другие. Таким образом, не читая инструкций, а просто пофантазировав или разглядывая названия ножек ЖКИ, можно догадаться о работе чужого устройства и всего мира в целом. Это один из способов пробить стену незнания и непонимания, особенно, если инструкцию переводил переводчик а не технарь.

Люди с опытом уже не фантазируют, а знают как работают аналогичные устройства. Но конечно, многое, созданное людьми попахивает нарушением логики и наличием ошибок. Ошибки есть у всех, но это не должно останавливать прогресс. Ошибки надо осознать, прощать и исправлять.

Итак посмотрим что за ножки у ЖКИ:

1. GND и Vcc питание ЖКИ. Подаем питание и уже ЖКИ работает, правда вместо цивильного теста пол экрана красится в черный цвет пол в белый. Стандарт не предусматривает ни какой тестовой информации а сделать ее было так легко.
2. V0 - напряжение от 0 до 5в - поляризация (яркость, контрастность) бывает что ярче когда 0, а бывает что когда 5в. По умолчанию притянуто резистором куда надо.
3. 8 ног данных. Мы не будем использовать все 8 ног, только 4, хотя и будем всегда передавать 8 бит информации за 2 раза. 8 ног это не экономично, это значит будет занято 8 ног от нашего процессора, а мы не можем быть такими расточительными.
4. RW - (1(5в)-чтение, 0-запись) читаем из ЖКИ или пишем в него. Конечно же мы будем только писать в ЖКИ и не будем контролировать, правильно ли мы вписали, как это предусмотрено стандартом. Мы будем четко и ясно отдавать команды без помех и надеяться что нас услышали. Все это ради экономии.
5. RS - (0-команда ЖКИ, 1-данные(буквы)) Как работает эта нога подробно описано в протоколе. Всю получаемую информацию ЖКИ воспринимает как команды и данные в зависимости от того какой сигнал на этой ноге. К командам относятся: сброс, очистка экрана, установка курсора, установка типа курсора. К данным относятся сам текст который рисуется на экране.
6. E - эта нога сигнализирует ЖКИ что данные/команду можно забирать с шины данных. Обычно такую управляющую ногу называют "строб". Такая нога всегда есть когда данные передаются по параллельной шине (одновременно 2-8 бит информации).
7. Катод и анод светодиодной подсветки.

Теперь когда мы описали ноги ЖКИ мы можем рассказать о самом низком уровне передачи данных из прерывания на ЖКИ.

Как мы говорили выше, есть у нас прерывание переполнения счетчика 0, которое настроено так, что оно срабатывает 7812,5 раз в сек. Если мы передаем 80 символов и каждый символ передаем за 2 вызова прерывания по 4 бита, то мы можем перерисовывать ЖКИ 7812,5/160 раз в секунду. Но это немного не так. На самом деле после передачи каждой буквы или команды иногда надо пропустить несколько вызовов прерывания, в зависимости от того, что мы передали, потому что ЖКИ надо дать время выполнить эту команду.

Существует 2 массива данных в моей программе:
V1[8] - низкие команды (понятные ЖКИ, из описания ЖКИ)
V2[80] - высокие команды (видеопамять) текстовое изображение ЖКИ.

Когда программа начинает передавать видеопамять в ЖКИ, она преобразует каждую высокую команду в несколько низких, понятных ЖКИ.

Вот самый низкий уровень передачи данных в ЖКИ. В этом кусочке программы происходит передача одной четверки (тетрады, половинки буквы или команды).

```
do{                          // Это одинарный цикл на весь ЖКИ для
                             // возможности в любой момент из него выпасть,
                             // используя break
if(fv.h)break;               // В данный момент посылка на ЖКИ не работает
  if(wV1){wV1--; break;}     // Если низкая задержка то ждем следующего
                             // вызова прерывания
  if(iV1<nV1)                // Если список низких команд не выполнен
    {
    x=V1[iV1];               // Это низкая команда для отражения ее на ногах
проца
    if(x&0x80){wV1=x<<1;}    // Если старший бит установлен, значит это
                             // низкая задержка
    else                     // Все остальные низкие команды
      {                      // превращаются в сигналы на ногах процессора
      sch_RW=bool(x&64);     // Устанавливаем режим чтение/запись
      sch_RS=bool(x&32);     // Команда или данные
      sch_DB4=bool(x&1);     // Младший бит квартета
      sch_DB5=bool(x&2);     // Средний бит квартета
      sch_DB6=bool(x&4);     // Средний бит квартета
      sch_DB7=bool(x&8);     // Старший бит квартета
      sch_E=bool(x&16);      // Фиксация данных (строб)
      }
    iV1++;                   // Переходим к следующей низкой команде
    if(iV1>=nV1){iV1=0; nV1=0;}// Если последняя низкая команда, все обнуляем
    }
  if(iV1<nV1)break;          // Если список низких команд не выполнен все
                             // остальные видеодела
                             // делать не надо в этом цикле вызова прерывания
```

**sch_RW, sch_DB4, sch_DB5, sch_DB6, sch_DB7, sch_E** - это название из схемы соответствующее ножке процессора и ножке ЖКИ. И при вписывании в этот бит 1 или 0 происходит появление +5в или 0в на соответствующей ноге в схеме.

Вот оно чудо! Программа меняет величину напряжения на ноге процессора. Информация управляет реальным миром! Дух управляет телом! Бог есть! Все остальное это детали.

Итак программист в разных местах программы обозначает какой шаблон он хочет видеть, подпрограмма рисования шаблонов рисует шаблон в видеопамяти, прерывание выведет видеопамять на ЖКИ. Для того чтобы экономить время процессора и ЖКИ предусмотрены механизмы не полной прорисовки шаблона, а только изменяемой части и не всех строчек разом, а только указанных. Прерывание и подпрограмма рисования шаблонов один раз написаны и отлажены, далее программист творчески манипулирует простым вызовом шаблонов в нужных местах программы. На этом мы закончим описывать механизм вывода информации на ЖКИ. Более детально смотрите сам текст программы.

## Урок №17 (структуры)

В нашем ЖКИ есть 2 канала зарядки. Так уж сложилось, что у процессора оказалось достаточно свободных ножек, чтобы подключить 2 канала, не привлекая дополнительной логики. И при выборе "2 канала" или "1 канал и балансир" выбор пал на первый вариант. Два абсолютно одинаковых канала управляются одной программой. Нужно написать подпрограмму работы с одним каналом и передать ей номер канала.

Одним выстрелом двух зайцев. Пишем одну подпрограмму, которую применим 2 раза. Разве это не халява?

Для того, чтобы это сработало, необходимо описать структуру каждого канала. В языке Си есть такое понятие "структура данных" - это удобный механизм описания

некоторого объекта, обладающего определенными свойствами. У нас таких объекта два, а структура у них одинаковая. Один раз описываем структуру и используем ее для 2х каналов.

Структура каждого из каналов - это все переменные, описывающие свойства канала, это все рычаги управления каналом, это все данные о канале, о его состоянии.

```
// Вот таким образом мы упрощенно описываем структуру канала
// Слово struct - это служебное слово Си
// Слово CHANNEL - это произвольное слово, теперь его можно
// использовать как тип для описания представителей структуры
struct CHANNEL
  {
  float I;
  float V;
  float T;
  };

// А в этом месте мы описываем наши два канала и каждый имеет одинаковую
// структуру как указано выше
CHANNEL Ch1, Ch2;

// А вот так мы будем работать со свойствами канала
Ch1.I=5;
Ch2.V=Ch1.V;
Ch1.T=GetTemperatur();
```

А вот описание настоящего канала:

```
struct CHANNEL// Структура, описывающая состояние канала в любой момент времени
{
BYTE A;                     // Номер подключенного аккумулятора из списка
long STime;                 // Время старта канала в десятых долях секунды (для вычисления полного времени работы канала)
long WTime;                 // Время работы канала в десятых долях секунды
long CTime;                 // Максимальное время выполнения одного цикла зарядки или разрядки в 1/10 сек.
long SecsT;                 // Для определения скорости роста температуры в каналах
BYTE Cycl;                  // Текущий цикл (загружается при пуске канала)
        // 0000 - зарядка
        // 0001 - разрядить и зарядить 1 раз
        // 0010 - разрядить и зарядить 2 раза
        // 0011 - разрядить и зарядить 3 раза
float k0, k1, k2;           // Коэффициенты для расчета тока зарядки
float kr0, kr1, kr2;        // Коэффициенты для расчета тока разрядки
float kv0, kv1, kv2;        // Коэффициенты подстройки напряжения
float Rch;                  // Сопротивление шунта зарядки
BYTE f1;                    // Флаги алгоритма зарядки 1 (10й байт типа аккума)
BYTE f2;                    // Флаги алгоритма зарядки 2 (11й байт типа аккума)
BYTE n;                     // Количество последовательных элементов в аккуме (Кол-во послед. банок)
WORD
ftPause:1,                  // Вторая попытка по перегреву
h:1,                        // Зарядка для хранения
Zaryd:1,                    // В настоящий момент вообщето вцелом идет заряд иначе разряд
LZaryd:1,                   // Локально сейчас идет заряд (true-заряд, false-разряд)
                            // Для отслеживания режимов декристаллизации
C:1,                        // 0-Стоп 1-Старт
dT:1,                       // 0-Скорость температуры < 2гр./мин. 1-больше (перегрев)
DP:1,                       // Динамический 0-Нет дельтапика 1-Есть дельта пик
DPs:1,                      // Статический 0-Нет дельтапика 1-Есть дельта пик
First:1,                    // 1-только что запустили 0-давно работаем
Stop:1;                     // 1-получен приказ на выключение 0-не получен
BYTE Speed;                 // 0-медленно 1-нормально 2-быстро
WORD Pause;                 // Необходима для стабилизации на тл494 заказанного тока в 1/10 сек.
BYTE Fasa;                  // Фаза заряда (0-нулевая, 1-основная, 2-капельный или струйный)
WORD WW;                    // Скорость роста и убывания тока изначально
WORD W;                     // Скорость роста и убывания тока может убывать при непревышении напряжения
WORD i;                     // Установленный ток (0-0xfffe) - величина ШИМ
WORD iDec;                  // Установленный ток (0-0xfffe) - величина ШИМ для локального процесса разрядки
BYTE sDec;                  // Стадия 0-Зарядка 1-Пауза 2-Разрядка 3-Пауза
BYTE pDec;                  // Переменная отсчета цикла декристаллизации от ch.PDes до нуля в 1/10 секундах
BYTE PDec;                  // Константа цикла декристаллизации из свойств типа аккума
BYTE TDec;                  // Константа цикла декристаллизации из свойств типа аккума
BYTE Imin, Imax;            // Плавное повышение тока в процессе зарядки
float I, V, T, Told;        // Реальные измеренные ток, напряжение и температура
float Tmax, TT;             // Абсолютная температура и скорость роста температуры перегрева
float Vst;                  // Напряжение статическое
float Vmax;                 // Используется при заряде для определения DP динамического
float Vmaxs;                // Используется при заряде для определения DP статического
float Vmin0;                // Минимальное напряжение для нулевой фазы
float Vmin1;                // Минимальное напряжение для первой фазы
float Vh;                   // Напряжение хранения
float dV;                   // Константа - величина дельтапика (во второй фазе используется для капильного заряда)
float II;                   // Принято решение установить такой ток
float IIDec;                // Принято решение установить такой ток для локального процесса разрядки
float VV;                   // Принято решение не превышать такое напряжение при заряде
float Z;                    // Емкость Ач
float InZ;                  // Полученный интегральный заряд Ач подсчитывается еже 1/10 секундно (при выводе надо поделить на
36000)
BYTE Msg;                   // Код остановки канала
         // 0 - Код не предусмотрен (канал не запускался)
         // 1 - Ток упал до нуля при заряде
         // 2 - Превышение лимита времени
         // 3 - Превышение температуры
         // 4 - Превышение скорости роста температуры
         // 5 - Превышение емкости заряда в 1.2 раза
         // 6 - Напряжение достигло минимума при разряде
         // 7 - Провисло питание до 11 вольт
         // 8 - Обнаружился дельтапик
         // 9 - На заряжаемом аккуме напряжение ниже минимального "это неправильно"
         // 10 - Зашкал тока
         // 11 - Зашкал напряжения
         // 12 - Остановлен пользователем
         // 13 - Напряжение ниже нулевой фазы
         // 14 - Напряжение достигло Vh хранения
         // 15 - Ошибки в настройках типов
         // 16 - Напряжение достигло нужного уровня при заряде
         // 17 - Перегрев схемы
}Ch1, Ch2;
```

А вот так в главном цикле вызывается обработка каждого канала. Ищите Ch1 и Ch2.

```
// Главная программа запускается по ресету
int main(void)
{
fc=255;
#include "init.cpp"        // Инициализация всего
#include "zagruzka.cpp"    // Загрузка начальных переменных
while(true)
  {
  wdt_reset();             // Сброс собаки
  #include "sh.cpp"        // Прорисовка шаблонов
  #include "test.cpp"      // Рассчитываем все переменные по каналам
  if(iK1!=iK0)             // Если есть необработанные кнопки рисуем меню
    {
    #include "menu.cpp"    // Обработка кнопок
    }
  TestMainParam();         // Если какой нибудь канал работает или тестирование
  if(Ch1.C)Go(Ch1);        // Если запущен канал 1
  if(Ch2.C)Go(Ch2);        // Если запущен канал 2
  #include "uart_in.cpp"   // Если надо чтонибудь получить с COM-порта
  #include "uart_out.cpp" // Если надо чтонибудь послать на COM-порт
  }
}
```

Как видите, в структуре канала есть переменные в которых отражен желаемый ток и желаемое напряжение. Канал не сразу реагирует и выставляет нужное напряжение и ток на аккумуляторе, а с определенной скоростью реакции. Кроме того, в настройках канала отражены текущие фазы зарядки. А также отражены результаты, достигнутые каналом. Как это все учесть?

Основная процедура управлением зарядкой/разрядкой Go() действует следующим образом:

1. Если это первый запуск Go, то загружаем в настройки канала все что мы знаем об этом аккумуляторе и его химии, обнуляем все что надо и ставим по умолчанию все что надо. Играем музыку старта.
2. Проверяем все экстренные проверки (зашкаливание, проверка на вшивость и т.д.)
3. Если проверки показали что надо остановиться, то останавливаемся и фиксируем причину. Играем музыку конца.
4. Если необходима задержка выскакиваем из программы столько раз, пока не истечет задержка, а вызывается Go() более 100 раз в секунду.
5. Обработка фазы декристаллизации, установка мелкопоместных задач по току, задержки опять ток разрядки или зарядки.
6. Опять длинная пауза по времени, которая может завершить Go() в этот раз.
7. Расчет всех токов и напряжений из данных АЦП. Проверка всех условий зарядки, разрядки, достижения требуемых напряжений и токов. Главная цель проверить не конец ли это.
8. Если закончилась очередная фаза тренировки, то пере инициализация для запуска нового цикла.
9. А вот теперь в зависимости от того что есть в наличии и того чего хочется и от допустимой скорости реагирования, выставляем новый ток через процедуру выставления тока.

Пока это все.