DI HALT

AVR. Учебный курс.

v. 1.1

Компиляция Вейселя Артема

СПб, 2012

AVR. Учебный курс. Постановка задачи

Этот пост можно смело считать кратким описанием того безобразия, что я называю AVR. Учебный Курс.

Сразу оговорюсь, что курс у меня краткий. Он не будет содержать в себе исчерпывающий материал по теме. Я не буду делать упор на подробности, потому что их все равно все не опишешь — полное описание на ту же ATMega16 занимает порядка 400 страниц убористого текста.

Моя основная цель — дать понять как это работает, как это использовать, как это настроить, чтобы получить хоть что-то рабочее. Поэтому я много уделяю алгоритмам, организации логики программ — классическим приемам программирования. Стараюсь не забывать и про оптимизацию.
Это, возможно, будет неинтересно тем кто учился на программистов, но вот электронщикам, изучающим программирование, это редко дают даже в ВУЗах.

Поэтому я дам лишь один-два самых ходовых примера по каждой теме, которые пригодятся в первую очередь, а остальное тебе придется додумывать по ходу пьесы самому. Впрочем, всегда можно будет спросить в комментариях.

О комментариях
Их обычно много, их пишут адекватные люди и в них часто можно найти массу дополнений и уточнений по моим статьям. А также ответы на многие смежные вопросы . Так что прежде чем что то спрашивать, рекомендую прочитать, хотя бы бегло, комменты. Возможно в них уже будет ответ на твой вопрос. Ну и отвечать я стараюсь на все комментарии к любому посту журнала.

О структуре курса
Курс начинается с ассемблера. Так как я считаю, что без понимания работы контроллера на уровне машинных инструкций невозможно научиться писать надежные и быстрые программы. Знание ассемблера позволяет в разы быстрей находить баги и ошибки в собственных программах. Да и вообще, на самом деле, ассемблер куда проще изучать чем тот же Си. Просто потому, что там все действия видны как на ладони. Достаточно только прогнать по ним отладчик AVR Studio.

В Си же есть ряд мощных заморочек с оптимизатором да и вообще, будучи более высокоуровневым, Си намного более сложен по структуре чем ассемблер. И изучать его с нуля сложней — т.к. в случае когда очевидная конструкция работает «не так» не знаешь куда бежать и за что хвататься. А если знаешь ассемблер, то открываешь дизасм, прогоняешь мутный участок и все баги выходят из сумрака.

После ассемблерной части сразу же начинается Сишный курс. Он написан так, как если бы все изучать с самого нуля. Даже без ассемблера, но к асму я там все равно постоянно обращаюсь. Так что рекомендую проходить все последовательно.

Некоторый материал может попасться раньше или позже чем он реально понадобиться, так что советую вдумчиво читать его по ходу обучения и бегло проглядеть от начала и до конца.

О ошибках
Я их совершаю, более того, я их совершаю часто. Могут быть как опечатки и технические неточности, так и откровенные ляпы :) Так что не стоит все написанное у меня рассматривать как истину в последней инстанции. Если что то не работает или работает не так как надо, то открывай даташит и сравнивай параметры — я мог и ошибиться. Если это так то сообщи мне в комментариях — исправлю. Если в чем то сомневаешься — спрашивай, разберемся.
В даташитах, кстати, тоже бывают ошибки. Более того, они бывают и в самих контроллерах на аппаратном уровне. Особенно в первых ревизиях. Об ошибках контроллеров и прошлых ревизий даташитов есть отдельный раздел. Называется ERRATA, находится в конце каждого даташита на контроллер. Советую внимательно его прочитать.

О языках
Вся техническая документация написана на английском. Но незнание языка — это не отмазка. Сейчас 21 век, а еще в конце 20го века, крошечная программка Сократ'97 умела вполне сносно переводить с английского на русский, не литературно, конечно, но понять суть можно было. За 13 лет технологии машинного перевода продвинулись весьма значительно =)))) Так что читайте мануалы.

О примерах
Примеры я даю в первой половине курса на ассемблере, во второй на Си и, если речь идет о алгоритме, в Си-подобном псевдокоде.

Однако, примеры просты и перегнать из Си в ассемблер и наоборот несложно. Особенно когда речь идет о инициализации и запуске в работу периферии.

Отмазки в стиле «Я знаю Си(ассемблер), а а вот ассемблер(си) не понимаю, есть пример на Си(асме)?» идут от нежелания чуть чуть подумать, внимательно посмотреть на то что и куда записывается :))) Не важно на каком языке что записано! Главное, что и в какие порты пихается, это понять можно с первого взгляда, на чем бы не писалось — на асме, си, паскале или бейсике.
Я, конечно, отвечаю и на такие вопросы. Но настроение мне это не повышает :)

О телепатии
Я ей не владею. Так что если что то не работает, то не надо мне пытаться съесть мозг вопросами. Куда лучше приложить тот код который не работает. Причем не стоит его копипастить в форму ввода комментариев — его весь перекосит, половина символов закосит под хтмл тэги, а читать все это вовсе будет невозможно. Да и проблема может быть вовсе и не там где тебе кажется.

Возьми и пихни в архив весь проект из AVR Studio и залей на какой-нибудь файлхостинг (их миллион сейчас. Тот же narod.yandex.ru [1]), а мне в комменты скинь ссылочку на архив и описание проблемы. Быстрей будет.

А раньше было же не так! Что случилось с старыми статьями?
А как собачки — сдохли и все. Старый курс был весьма коряв, в нем содержались мощные дыры, а также не было единой отладочной платформы. Сейчас, с появлением демоплатыPinboard [2] эта проблема решена.

Плюс, за два года его существования, под градом вопросов и комментариев, была заново переосмыслена структура курса. Посему старые статьи были подвергнуты жесткой редактуре под новые реалии. Думаю, так стало лучше и логичней.

Чего не будет в курсе
У меня гарантированно не будет ботвы вида «смотрите как легко и просто мигнуть диодиком», «скопируйте эту строку в свою программу и на LCD увидите текст. Это просто». Я не буду показывать как писать программы для AVR на примере какого-нибудь, нафиг тебе не нужного устройства, вроде частотомера или музыкального звонка. Захочешь частотомер — сам потом сделаешь, уже без моей помощи :)

Что будет в курсе
В курсе будут куски типовых решений, ассемблерные и сишные процедуры. Будет подробный разбор как работает и как сделать ту или иную конструкцию, такие как цикл, флаговый автомат, табличный переход и тыды. Как организовать многопоточное выполнение кода и псевдомногозадачность. Как оседлать прерывания и всякие извраты с кодом. Как зажмотить два три байта и как выйграть два три такта. Как взорвать мозг тому, кто захочет разобраться в твоем коде и как сделать так, чтобы все было логично и просто.

Разумеется, я покажу и как дидом мигнуть, и как на LCD экран вывести что-то. А также множество других не менее интересных приемов. Но будет это далеко не сразу, а лишь тогда, когда выданной тебе инфы будет достаточно для представления себе полной картины происходящего действа.

Вплоть до того как и в каком порядке перемещаются байты по регистрам. Поэтому первым делом будет разобрана работа ядра контроллера. Как и в каком порядке он щелкает байты, а уж потом как дрыгает ножками.

И ты удивишься тому, какой же простой и логичный язык Ассемблера. А потом, без проблем, пересядешь на Си. Освоив, тем самым надфиль и кувалду микроконтроллерного программирования. Я в этом уверен.

А если что, то я всегда рядом, стоит только оставить комментарий… =))))

З.Ы.
Курс находится в стадии непрерывного развития и усовершенствования, статьи дополняются и перетасовываются для получения более логичной и последовательной структуры. Причем делается это втихую, без уведомления. Рекомендую переодически проглядывать курс с начала и до конца, возможны изменения.

AVR. Учебный Курс. Вводная. Что такое микроконтроллер.

Микроконтроллер это, можно сказать, маленький компьютер. Который имеет свой центральный процессор (**регистры, блок управление и арифметическо-логическое устройство**), **память**, а также разную **периферию**, вроде **портов ввода вывода**, таймеров, контроллеров прерываний, генераторов разных импульсов и даже аналоговых преобразователей. Всего не перечислишь. Как нельзя перечислить все применения микроконтроллеров.

Но, если сильно все упростить, то основной функцией микроконтроллера является «дрыганье ножками». Т.е. у него есть несколько выводов (от 6 до нескольких десятков в зависимости от модели) и на этих выводах он может выставить либо 1 (высокий уровень напряжения, например +5вольт), либо 0 (низкий уровень напряжения, около 0.1 вольта) в зависимости от программного алгоритма зашитого в его память. Также микроконтроллер может определять состояние сигнала на своих ножках (для этого они должны быть настроены на вход) — высокое там напряжение или низкое (ноль или единица). Современные микроконтроллеры также почти поголовно имеют на борту Аналогово Цифровой Преобразователь — это штука подобная вольтметру, позволяет не просто отследить 0 или 1 на входе, а полноценно замерить напряжение от 0 до опорного (обычно опорное равно напряжению питания) и представить его в виде числа от 0 до 1024 (или 255, в зависимости от разрядности АЦП)

Из него можно сделать и умный дом, и мозги для домашнего робота, систему интеллектуального управления аквариумом или просто красивое светодиодное табло с бегущим текстом. Среди электронных компонентов МК это один из самых универсальных устройств. Я, например, при разработке очередного устройства предпочитаю не заморачиваться на различного рода схемотехнические извраты, а подключить все входы и выходы к микроконтроллеру, а всю логику работы сделать программно. Резко экономит и время и деньги, а значит деньги в квадрате.

Микроконтроллеров существует очень и очень много. Практически каждая уважающая себя фирма по производству радиокомпонентов выпускает свой собственный контроллер. Однако и в этом многообразии есть порядок. МК делятся на семейства, все их я не перечислю, но опишу лишь самые основные восьмиразрядные семейства.

MSC-51
Самое обширное и развитое это **MSC-51**, старейшее из всех, идущее от **intel 8051** и ныне выпускаемое массой фирм. Иногда кратко зовется **C51**. Это 8-ми разрядная архитектура, отличается от большинства других восьмиразрядников тем, что это **CISC** архитектура. Т.е. одной командой порой можно совершить довольно сложное действие, но команды выполняются за большое число тактов (обычно за 12 или 24 такта, в зависимости от типа команды), имеют разную длину и их много, на все случаи жизни. Среди контроллеров архитектуры **MSC-51** встречаются как динозавры вроде **AT89C51**, имеющие минимум периферии, крошечную память и неважнецкое быстродействие, так и монстры вроде продукции **Silicon Laboratories** имеющие на борту весьма мясистый фарш из разнокалиберной периферии, огромные закрома оперативной и постоянной памяти, мощные интерфейсы от простого **UART**`а до **USB** и **CAN**, а также зверски **быстрое ядро**, выдающее до 100 миллионов операций в секунду. Что касается лично меня, то я обожаю архитектуру C51 за ее чертовски приятный ассемблер на котором просто кайфово писать. Под эту архитектуру уже написаны гигабайты кода, созданы все мыслимые и немыслимые алгоритмы.

Atmel AVR
Вторым моим любимым семейством является **AVR** от компании **Atmel**. Вообще **Atmel** производит и **MSC-51** контроллеры, но все же основной упор они делают на **AVR**. Эти контроллеры уже имеют 8-ми разрядную **RISC** архитектуру и выполняют одну команду за один такт, но в отличии от классического **RISC** ядра имеют весьма развесистую систему команд, впрочем не такую удобную как у C51, за что я их недолюбливаю. Но зато **AVR** всегда снаряжены как на войну и просто напичканы разной периферией, особенно контроллеры подсемейства **ATMega**. А еще их очень легко прошивать, для этого не нужны ни специализированные программаторы, ни какое либо другое сложное обрудование. Достаточно лишь пяти проводков и компьютера с **LPT** портом. Простота освоения позволила этому контроллеру прочно запасть в сердца многих и многих радиолюбителей по всему миру.

Microchip PIC.
Еще один 8-ми разрядный **RISC** микроконтроллер, отличается весьма извратской системой команд, состоящей всего из пары десятков команд. Каждая команда выполняется за четыре такта. есть ряд достоинств, в первую очередь это низкое энергопотребление, и быстрый старт. В среднем **PIC** контроллере нет такого количества периферии как в AVR, но зато самих модификаций **PIC** контроллеров существует такое количество, что всегда можно подобрать себе кристалл с периферией подходящей точно под задачу, не больше не меньше. На **PIC**`ах традиционно построены бортовые компьютеры автомобилей, а также многочисленные бытовые сигнализации.

Какое же семейство выбрать? О, это сложный вопрос. На многочисленных форумах и конференциях по сей день идут ожесточенные бои на тему какое семейство лучше, фанаты **AVR** грызутся с приверженцами **MSC-51**, попутно не забывая пинать по почкам **PIC**`овцев, на что те отвечают тем же.

Ситуация тут как в Starcraft :) Кто круче? Люди? Зерги? Протоссы? Все дело в применении, масштабах задач и массе других параметров. У каждого семейства есть свои достоинства и недостатки. Но лично я бы выбрал AVR и вот по каким причинам:

- 1. Доступность в России. Эти контроллеры заслуженно популярны и любимы народом, а значит наши торговцы их охотно возят. Впрочем, как и PIC. С MSC-51 ситуация хуже. Морально устаревшие AT89C51 достать не проблема, но кому они нужны? А вот современные силабы это уже эксклюзив.
- 2. Низкая цена. Вообще низкой ценой в мире славится PIC, но вот ирония — халявы начинаются только если брать его вагонами. На деле же, на реальном прилавке, AVR будет процентов на 30-40 дешевле чем PIC при несколько большем функционале. С MSC-51 ситуация ясна еще по первому пункту. Эксклюзив это не только редко, но и дорого.
- 3. Очень много периферии сразу. Для серийного устройства это скорей недостаток. Куда лучше иметь только то, что надо в текущей задаче, а остальное чтобы не мешалось и не кушало зря энергию. Этим славится PIC со своим развесистым модельным рядом, где можно найти контроллер в котором будет нужное и не будет ненужного. Но мы то собираемся изучать и делать для себя! Так что нам лучше чтобы все, сразу и про запас. И вот тут AVR на голову выше чем PIC, выкатывая раз за разом все более фаршированные контроллеры. Купил себе какую-нибудь AtMega16A и все, можешь все семейство изучить.
- 4. Единое ядро. Дело в том, что у всех современных AVR одинаковое ядро с единой системой команд. Есть лишь некоторые различия на уровне периферии (и те незначительные). Т.е. код из какой нибудь крошечной ATTiny13 легко копипастом перетаскивается в ATMega64 и работает почти без переделок. И почти без ограничений наоборот. Правда у старых моделей AVR (всякие AT90S1200) совместимость сверху вниз ограниченная — у них чуть меньше система команд. Но вот вверх на ура. У Микрочипа же существует целая куча семейств. PIC12/16/18 с разной системой команд. 12е семейство это обычно мелочь малоногая (вроде Tiny в AVR), а 18 это уже более серьезные контроллеры (аналог Mega AVR) И если код с 12го можно перетащить на 18, то обратно фиг.
- 5. Обширная система команд контроллеров AVR. У AVR около 130 команд, а у Microchip PIC всего 35. Казалось бы PIC в выйгрыше — меньше команд, проще изучить. Ну да, именно так и звучит микрочиповский слоган, что то вроде «Всего 35 команд!». Только это на самом деле фигня. Ведь что такое команда процессора? Это инструмент! Вот представь себе два калькулятора — обычный, бухгалтерский и инженерный. Бухгалтерский куда проще изучить чем инженерный. Но вот попробуй посчитать на нем синус? Или логарифм? Нет, можно, не спорю, но сколько нажатий кнопок и промежуточных вычислений это займет? То то же! Куда удобней работать когда у тебя под рукой куча разных действий. Поэтому, чем больше система команд тем лучше.
- 6. Наличие бесплатных кроссплатформенных компиляторов Си. Конечно, кряк всегда найти можно. Где где, а в нашей стране это проблемой никогда не было. Но зачем что то воровать если есть халявное? ;)
- 7. Ну и последний аргумент, обычно самый весомый. Наличие того, кто бы научил и подсказал. Помог советом и направил на путь истинный. Я выбрал для себя AVR и на этом сайте (по крайней мере пока) досконально будет разбираться именно это семейство, а значит выбора у тебя особого нет :))))))

Ой, но этих же AVR целая прорва. Какой взять???
Интересный вопрос. Вообще МК лучше выбирать под задачу. Но для изучения лучше хапнуть что то фаршированное.

Для начала разберем маркировку, чтобы ты по прайсу сразу мог понять что за зверь перед тобой. Вот тебе пример

ATmega16A — 16PI

- **AT** — сделано в Atmel
- **Mega** — вид семейства. Существует еще Tiny и Xmega (новая — фаршу жуть, полный вертолет). Вообще задумывалось, что Тини это, вроде как, малобюджетное с малым количеством фарша и вообще ущербная, а Мега наоборот — все и сразу. В реальности, разница между семействами Тини и Мега по фаршу сейчас минимальная, но в Тини меньше памяти и корпуса у нее бывают с числом выводов от 6 до 20.
- **16** — количество памяти флеша в килобайтах. Вообще тут не все так просто. Числом памяти является степень двойки. Так что Mega162 это не контроллер со 162КБ флеша, а своеобразная Мега16 модификации2 с памятью 16кб. Или вот Мега88 — не 88кб, а 8кб флеша, а вторая 8 это вроде как намек на то, что это дальнейшее развитие Мега8. Аналогично и Мега48 или Мега168. Тоже самое и семейством Тини. Например, Тини2313 — 2килобайта флеша. А что такое 313? А хрен знает что они имели ввиду :) Или Тини12 — 1кб Флеша. В общем, фишку просек.
- **A** — префикс энергопотребления (обычно). Этой буквы может и не быть, но в новых сериях она присутствует почти везде. Например, V и L серии — низковольтные, могут работать от 2,7 вольт. Правда за низковольтность приходится платить меньше частотой. Но оверклокинг возможен и тут, ничто человеческое нам не чуждо :) A и P имеют новые серии AVR с технологией PicoPower т.е. ультраэкономичные. Разницы по фаршу и внутренней структуре с их безиндексовыми моделями нет, тут

все различие в работе всяких спящих режимов и энергопотреблении. Т.е. Mega16A легко меняется на Mega16 без A. И ничего больше менять не нужно.
- **16** — Предельная тактовая частота в мегагерцах. В реальности можно разогнать и до 20 ;)
- **P** — тип корпуса. Важная особенность. Дело в том, что далеко не всякий корпус можно запаять в домашних условиях без геморроя. Рекомендую пока обратить внимание на P — DIP корпус. Это громоздкий монстр, но его легко запаять, а, главное, он легко втыкается в специальную панельку и вынимается из нее обратно. Корпуса вида SOIC (индекс S) или TQFP (индекс A) пока лучше отложи в сторонку. Без хорошего опыта пайки и умения вытравить качественную печатную плату к ним лучше не соваться.
- **I** — Тип лужения выводов. I — свинцовый припой. U — безсцвиновый. Для тебя никакой совершенно разницы. Бери тот что дешевле.

Рекоменую следующие модели:

- ATMega16A-16PU — недорогой (около 100-150р), много выводов, много периферии. Доступен в разных корпусах. Прост, под него заточен мой учебный курс и все дальнейшие примеры.
- ATTiny2313-20SU — идеальный вариант для изготовления всяких часов/будильников и прочей мелкой домашней автоматики. Дешев (рублей 40), компактен. Из минусов — нет АЦП.
- ATmega48/88/168 любой из этих контроллеров. Компактен (в корпусе tqfp является самым тонким и мелким из AVR), дешев (рублей 100-150), фаршированн донельзя.
- ATmega128 для искушенных. Большой, мощный, дофига памяти. Дорогой (около 400р)

C vs Assembler

Если ты впервые столкнулся с микроконтроллерами, то наверняка у тебя стал выбор на чем писать.

На **Си** или на **Ассемблере**. Выбор не прост, не зря программисты однокристальщики раскололись на два непримиримых лагеря. Одни с пеной у рта доказывают, что те кто пишут на Си лохи изнеженные, а настоящий брутальный кодер должен воспринимать только ассемблер. Другие же, усераясь, доказывают, что на ассемблере разве что лампочками помигать можно, а какой либо серьезный проект делать на низком уровне невозможно.

Но, истинна, как всегда, находится посредине. **Каждый уважающий себя программер должен знать ассемблер**, а вот писать должен **на том, что более подходит под масштаб задачи**. Особенно это касается микроконтроллеров.

Assembler+
Парадоксально, но под микроконтроллеры писать на ассемблере ничуть не сложней чем на Си. В самом деле, программирование контроллера неотделимо от его железа. А когда пишешь на ассемблере, то обращаешься со всей периферией контроллера напрямую, видишь что происходит и как это происходит. Главное, что при этом **ты четко понимаешь что, как и зачем ты делаешь**. Очень легко отлаживать и работа программы как на ладони. Да и сам ассемблер изучается очень быстро, за считанные дни. Достаточно постоянно иметь перед глазами систему команд и навык программирования алгоритмов. А соответствующее состояние мозга, когда начинаешь мыслить ассемблерными конструкциями наступает очень быстро.

C-
А вот с высокими языками, например с Си, ситуация уже куда хуже. Да, повторить какой нибудь пример из обучалки, вроде вывода строки на дисплей, Сях куда проще, подключил нужную библиотеку, набрал стандартную команду и вот тебе результат. Но это только так кажется. На самом деле, стоит чуть вильнуть в сторону, как начинается темный лес. Особенно когда дело начинает касаться адресации разных видов памяти,

Напомню, что у AVR, как и подавляющего числа МК, память разделена на код и данные и адреса у них в разных адресных пространствах о которых стандарт Си ничего не знает, ведь он расчитан на некий универсальный вычислитель. А тут приходится вводить всякие модификаторы, квалификаторы и без яростного курения мануалов разобраться в этих примочках бывает очень сложно. Особенно работа с памятью осложняется тем, что программа то работает в том и другом случае, но вот если неправильно указать тип памяти, то оперативка забьется константами и в дальнейшем программу скрючит в самый неподходящий момент. Вот и найди такой глюк

работе на прерываниях, использованию многопоточных процессов, использующих одни и те же ресурсы.

Вот и получается, что программа вроде компилится, но работает совершенно не так как тебе нужно, а почему непонятно. Или мусор начинает вываливать вместо данных, или перезагружается невпопад. Или работает, но

иногда чудит. Пытаешься смотреть что происходит в системных регистрах и переменных, а там каша какая то. Пытаешься понять как это получилось и тупо загниваешь, т.к. не можешь найти концы.

В таких случаях обычно открывают листинг и смотрят что там происходит в коде на уровне команд процессора, но вот засада — без знания ассемблера там делать нечего! Зато если понимаешь то как работает программа на уровне команд, то найти багу вроде срыва стека или нарушения атомарного доступа проще простого.

Ну и такие гадкие вещи как **overhead**. Программа на Си требует значительно больше оперативной памяти, значительно больше **места во flash памяти**, работает куда медленней чем ассемблерная программа. И если на обычном компьютере с его гигагерцами частот, гигабайтами оперативки и дисковой памяти это не столь критично, то вот для контроллера с жалкими килобайтами флеша, у которого частота порой не превышает 16 мегагерц, а оперативки и килобайта не наберется, такой расход ресурсов более чем критичен.

Кроме того существуют такие контроллеры как **ATTiny 1x** у которых либо вообще нет оперативки, либо она такая мизерная, что даже стек там сделан аппаратным. Так что на Си там ничего написать в принципе нельзя.

Assembler+
Представь, что ты прораб, а компилятор это банда джамшутов. И вот надо проделать дырку в стене. Даешь ты джамшутам отвертку и говоришь — ковыряйте. Проковряют они отверткой бетонную стену? Конечно проковыряют, вопрос лишь времени и прочности отвертки. Отвертка сточится (читай памяти процессора или быстродействия не хватит)? Не беда — дадим джамшутам отвертку побольше, благо разница в цене между большой отверткой и маленькой копеечная. В самом деле, зачем прорабу, руководителю, знать такие низкоуровневые тонкости, как прочность и толщина бетонной стены, типы инструмента. Главное дать задание и проконтроллировать выполнение, а джамшуты все сделают сами.
Задача решается? Да! Эффективно это решение? Совершенно нет! А почему? А потому что прораб не знал, что бетон твердый и отверткой его проковырять сложно. А будь прораб сам когда то рабочим, пусть даже не профи, но своими руками положил плитку, посверлил дырки, то впредь таких идиотских заданий бы не давал. Конечно, нашего прораба можно и в шараге выучить, дав ему всю теорию строения стен, инструмента, материалов. Но ты представь сколько это сухой теории придется перелопатить, чтобы чутье было интуитивным, на уровне спинного мозга? Проще дать в руки инструмент и отправить сверлить стены. Практика — лучший учитель.

Также и с ассемблером. Хочешь писать эффективные программы на высокоуровневом языке — изучи хотя бы один ассемблер, попиши на нем немного. Чтобы потом, глядя на любую Сишную строку, представлять себе во что это в итоге компилируется и как обрабатывается контроллером. Очень помогает в отладке и написании, а уж про ревесирование чужих программ я вообще не говорю.

Assembler-
Но засиживаться в ассемблерщиках и стараться все сделать на нем далеко не обязательно. Ассемблерный код тяжелей переносить с контроллера на контроллер, в нем при переносе можно наделать много незначительных, но тем не менее фатальных ошибок. Отладка которых занимает много времени. Большие проекты писать на ассемблере то еще развлечение. На ассемблере трудно сделать полноценную библиотеку, подключаемую куда угодно. Ассемблер жестко привязан к конкретному семейству контроллеров.

C+
Си хорош за счет огромного числа готового кода, который можно очень легко и удобно подключать и использовать в своих нуждах. За большую читабельность алгоритмов. За возможность взять и перетащить код, например, с AVR на ARM без особых заморочек. Или с AVR на PIC. Разумеется для этого надо уметь ПРАВИЛЬНО писать на Си, выделяя все аппаратно зависимые части в HAL.

Общий расклад примерно таков, что использование высокоуровневого языка на контроллерах с обьемом памяти меньше 8 килобайт является избыточным. Тут эффективней писать все на Ассемблере. Особенно если проект подразумевает не просто мигание светодиодом.
8-16 килобайт тут уже зависит от задачи, а вот пытаться писать на ассемблере прошивку более 16 килобайт можно, но это напоминает прокладку тоннеля под Ла Маншем с помощью зубила.

В общем, знать надо и то и другое. Настоятельно тебе рекомендую начать изучать МК с ассемблера. А как только поймешь, что на асме можешь реализовать все что угодно, любой алгоритм. Когда досконально прочувствуешь работу стека, прерываний, организацию переходов и ветвлений. Когда разные трюки и хитрости, вроде игр с адресами возврата из прерываний и процедур, переходами и конечными автоматами на таблицах и всякие извраты будут вызывать лишь интерес, но никак не взрыв мозга из серии «Аааа как это работает??? Не понимаю?!!»
Вот тогда и можно изучать Си. Причем, изучать его с дебагером в руках. Не просто изучить синтаксис (там то как раз все элементарно), а понять ЧТО и КАК делает компилятор из твоего исходника. Поржать над его тупостью или наоборот поудивляться извратам искуственного интелекта. Понять как компилятор делает ветвления, как

организует циклы, как идет работа с разными типами данных, как ведется оптимизация. Где ему лучше помочь, написав в ассемблерном стиле, а где не критично и можно во всю ширь использовать языковые возможности Си.

А вот начать изучение ассемблера после Си мало кому удается. Си расслабляет, становится лень и впадлу. Скомпилировалось? Работает? Ну и ладно. А то что там быдлокод, та пофигу… =)

А как же бейсик, паскаль и прочие языки? Они тоже есть на AVR?
Конечно есть, например BascomAVR или MicroPASCAL и во многих случаях там все проще и приятней. Не стоит прельщаться видимой простотой. Она же обернется тем, что потом все равно придется переходить на Си.

Дело в том, что мир микроконтроллеров далеко не ограничивается одним семейством. Постоянно появляются новые виды контроллеров, развиваются новые семейства. Ведь кроме AVR есть еще и ARM, PIC, STM8 и еще куча прекрасных контроллеров со своими плюсами.
И под каждый из этих семейств есть Си компилятор. Ведь Си это, по сути, промышленный стандарт. Он есть везде и контроллер который не имеет под него компилятора популярным у профессионалов не станет никогда.

А вот на бейсик с паскалем, обычно, всем пофигу. Если на AVR и PIC эти компиляторы и сделали, то лишь потому, что процы эти стали особо популярны у любителей и там наверняка найдется тот, кто заинтересуется и бейсиками всякими. С другим семейством контроллеров далеко не факт, что будет также радужно. Например под STM8 или Cortex M3 я видел Pascal в лучшем случае только в виде кривых студенческих поделок. Никак не тянущих на нормальный компилятор.

Такой разный Си
С Си тоже не все гладко. Тут следует избегать компиляторов придумывающих свои диалектные фишки. Например, CodeVision AVR (CVAVR) позволяет обращаться к битам порта с помощью такого кода:

```
1 PORTB.7 = 1;            // Выставить бит 7 порта B в 1
```

Удобно? Конечно удобно! Вот только в Си так делать нельзя, стандарт языка не позволяет. И ни один другой Си компилятор такой кусок кода не примет. Т.к. по стандарту корректней будет так:

```
1 PORTB |= 1<<7;
```

Использование диалектов не позволит тебе скакать с компилятора на компилятор. Таскать код повсюду копипастом. Привязывает к одному конкретному компилятору и далеко не факт, что он окажется хорошим и будет поддерживать все новые контроллеры семейства.

Приплюснутый
Некоторое время назад я считал, что C++ в программировании микроконтроллеров не место. Слишком большой overhead. С тех пор мое мнение несколько поменялось.
Мне показали очень красивый кусок кода на C++, который компилился вообще во что то феерическое. Компактней и быстрей я бы и на ассемблере не факт что написал. А уж про читабельность и конфигурируемость и говорить не приходится. Все из знакомых программистов, кто видел этот код, говорили что-то вроде «Черт, а я то думал, что я знаю C++».

Так что писать на C++ можно! Но сделать компактный и быстрый код на C++ чертовски виртуозная задача, надо знать этот самый C++ в совершенстве. Судя по тому, что столь качественных примеров эффективности C++ при программировании под МК мне попадалось раз-два, то видимо пороговый уровень входа в эту область весьма и весьма велик.

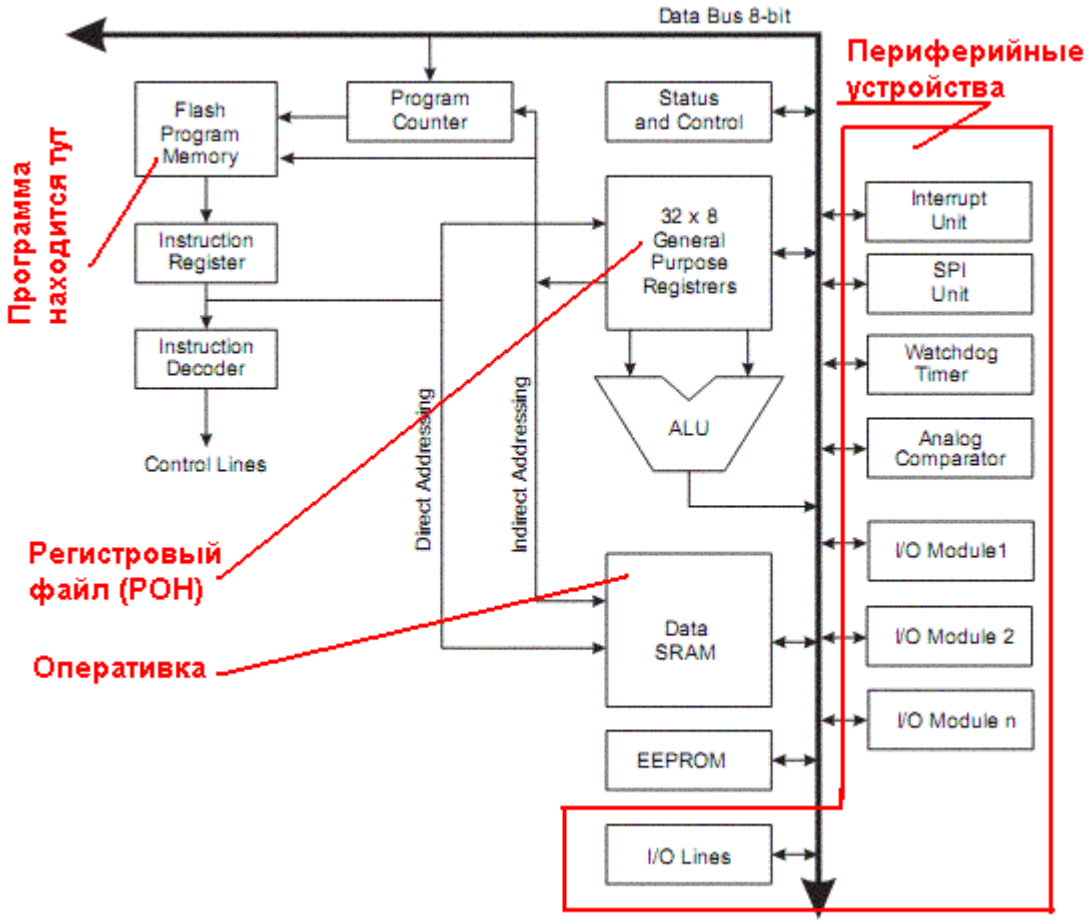
В общем, как говорил Джон Кармак, «хороший C++ код лучше чем хороший C код. Но плохой C++ может быть намного ужасней чем плохой C код».

AVR. Учебный курс. Архитектура.

Итак, камрад, прежде чем ты начнешь работать с контроллером, то неплохо бы тебе узнать что у него внутри. Поэтому дам тебе краткий ликбез по **архитектуре контроллера AVR**.

Основой любого микроконтроллера является вычислительное ядро. Во всех моделях AVR оно практически одинаковое и это большой плюс. Именно единство архитектуры обеспечивает легкую переносимость кода.

Итак, что же у нас в основе микроконтроллера, взгляни на диаграмму:



Ядро состоит, в первую очередь, из памяти программ (Flash Programm Memory) и Арифметико-логического устройства (ALU), блока управления (на диаграмме не показан) и программного счетчик (Program Counter). Также есть тактовый генератор, задающий импульсы относительно которых работают блоки микроконтроллера. Тактовый генератор можно сравнить с маятником и собачкой в будильнике: маятник туда сюда, собачка тикает по одному зубчику — шестеренки крутятся. Встала собачка — встал весь будильник.

При старте микроконтроллера значение программного счетчика равно 0000 — это адрес первой команды в нашей flash ROM. Микроконтроллер хватает оттуда два байта (код команды и ее аргументы) и отдает на выполнение в декодер команд (Instruction Decoder).

А дальнейшая судьба зависит от команды. Если это просто команда работы с какими-либо действиями, то они будут выполнены, а на следующем такте значение программного счетчика будет увеличено и из следующей пары ячеек памяти будут взяты еще два байта команды и также отправлены на выполнение.

Все интересней становится когда встречается команда перехода. В этом случае в Программный счетчик загружается адрес указанный в команде (абсолютный переход) или его значение увеличивается не на 1, а на столько сколько нужно и на следующем такте микроконтроллер возьмет команду уже с нового адреса.

Декодер команд загребает команду и скармливает ее логике блока управления, который уже пинает все остальные блоки, заставляя их делать нужные действия в нужном порядке.

Вся математика и обработка делается посредством ALU. Это, своего рода, калькулятор. Он может складывать, вычитать, сравнивать, сдвигать разными способами, иногда делить и умножать (это считается круто, встречается редко).

В качестве промежуточных операндов используются 32 ячейки — Оперативные регистры общего назначения РОН. Доступ к этим ячейкам самый быстрый, а число операций с их содержимым наиболее богатое. В ассемблере регистры эти называются просто R0,R1,R2 … R31. Причем делятся они на три группы:

Младшие R0..R15
Обычные регистры общего назначения, но какие то ущербные. С ними не работают многие команды, например, такие как загрузка непосредственного числа. Т.е. нельзя, например, взять и присвоить регистру число. Зато можно скопировать число из любого другого регистра.

Старшие R16..R31
Полноценные регистры, работающие со всеми командами без исключения.

Индексные R26...R31
Шесть последних регистров из старшей группы особенные. В принципе, их можно юзать и как обычные регистры общего назначения. Но, кроме этого, они могут образовывать регистровые пары X(R26:R27), Y(R28,R29), Z(R30:R31) которые используются как указатели при работе с памятью.

ОЗУ
Кроме 32 регистров в микроконтроллере есть оперативная память. Правда не везде — в младших семействах AVR Tiny12 и Tiny11 оперативной памяти нет, так что приходиться вертеться в 32 ячейках.

Оперативная память это несколько сотен ячеек памяти. От 64 байт до 4килобайт, в зависимости от модели. В этих ячейках могут храниться любые данные, а доступ к ним осуществляется через команды Load и Store.

То есть нельзя взять, например, и прибавить к ячейке в памяти, скажем, единицу. Нам сначала сделать операцию Load из ОЗУ в РОН, потом в регистре прибавить нашу единицу и операцией Store сохранить ее обратно в память. Только так.

EEPROM
Долговременная память. Память которая не пропадает после выключения питания. Если Flash может содержать только код и константы, а писать в нее при выполнении ничего нельзя (Это память Read Only), то в ЕЕПРОМ можно сколько угодно писать и читать. Но в качестве оперативки ее особо не поюзаешь. Дело в том, что цикл записи в EEPROM длится очень долго — миллисекунды. Чтение тоже не ахти какое быстрое. Да и число циклов перезаписи всего 100 000, что не очень много в масштабах работы оперативной памяти. ЕЕПРОМ используется для сохранения всяких настроек, предустановок, собранных данных и прочего барахла, что может потребоваться после включения питания и в основном на чтение. Эта память есть не во всех моделях AVR, но в подавляющем их большинстве.

Периферия
Периферия это внутренний фарш микроконтроллера. То что делает его таким универсальным. ALU, RAM, FLASH и Блок управления это как в компе Мать, Проц, Память, Винт — то без чего комп даже не запустится толком. То периферия это уже как сетевуха, видяха, звуковая карта и прочие прибамбасы. Они могут быть разными, разной степени крутости и навороченности, а также комбинироваться в разном порядке.

Именно по наличию на кристалле той или иной периферии происходит выбор микроконтроллера под задачу.

Периферии всякой придумано великое множество, всего я наверное даже не опишу. Но дам основной набор присутствующий почти во всех AVR, а также в других современных контроллерах.

- Порты ввода вывода — то без чего невозможно взаимодействие контроллера с внешним миром. Именно порты обеспечивают то самое «ножкодрыгательство» управляющее другими элементами схемы. Захотели получить на выводе единичку, дали приказ соответствующему порту — получите, распишитесь. Захотели узнать какой там сигнал на входе? Спросили у соответствующего порта — получили. Почти все выводы микроконтроллера могут работать в режиме портов ввода-вывода.
- UART/USART приемопередатчик — последовательный порт. Работает по тому же асинхронному протоколу что и древние диалапные модемы. Старый как мир, надежный и простой как кувалда. Подходит для связи с компьютером и другими контроллерами.
- Таймеры/счетчики — задача таймеров отсчитывать тики. Сказал ему отсчитать 100 тактов процессора — он приступит и как досчитает подаст сигнал. Им же можно подсчитывать длительность входных сигналов, подсчитывать число входных импульсов. Да много чего умеет таймер, особенно в AVR. Подробное описание функций таймера занимает добрых три десятка страниц в даташите. При том, что таймеров самих существует несколько видов и фарш у них разный.

- АЦП — аналоговый вход. Есть не у всех микроконтроллеров, но вещь полезная. Позволяет взять и замерить аналоговый сигнал. АЦП это своеобразный вольтметр.
- I2C(TWI) интерфейс — последовательная шина IIC. Через нее осуществляется связь с другими устройствами. На IIC можно организовать своеобразную локальную сеть из микроконтроллеров в пределах одного устройства.
- SPI — еще один последовательный протокол, похожа на IIC, но не позволяет организовывать сети. Работает только в режиме Мастер-Ведомый. Зато ОЧЕНЬ быстрая.
- Аналоговый Компаратор — еще один аналоговый интерфейс. Но, в отличии от АЦП, он не замеряет, а сравнивает два аналоговых сигнала, выдавая результат A>B или A<B в двоичном виде.
- JTAG/DebugWire — средство отладки, позволяет заглянуть в мозги контроллера с помощью специального адаптера, например такого, какой встроен в мою демоплату Pinboard [1]. Иной раз без него как без рук.
- PWM — ШИМ генератор. Вообще это не отдельный блок, а дополнительная функция таймера, но тоже полезная. С помощью ШИМ генератора легко задать аналоговый сигнал. Например, менять яркость свечения светодиода или скорость вращения двигателя. Да мало ли куда его применить можно. Число каналов ШИМ разное от контроллера к контроллеру.

Еще бывают встроенные USB, Ethernet интерфейсы, часы реального времени, контроллеры ЖКИ дисплеев. Да чего там только нет, моделей микроконтроллеров столько, что задолбаешься только перечислять.

Взаимодействие ядра с периферией
Ядро одно на всех, периферия разная. Общение между ними происходит через память. Т.е. у периферии есть свои ячейки памяти — регистры периферии. У каждого периферийного устройства их не по одной штуки. В этих регистрах находятся биты конфигурации. В зависимости от того как эти биты выставлены в таком режиме и работает периферийное устройство. В эти же регистры нужно записывать данные которые мы хотим выдать, например, по последовательному порту, или считывать данные которые обработал АЦП. Для работы с периферией есть специальные команды IN и OUT для чтения из периферии в регистр РОН и записи из регистра РОН в периферию соответственно.

Поскольку ядро одинаковое, а периферия разная, то при переносе кода на другую модель микроконтроллера надо только подправить эти обращения так как название периферийных регистров от модели к модели может чуток отличаться. Например, если в контроллере один приемопередатчик UART то регистр прием данных зовется UDR, а если два, то у нас есть уже UDR0 и UDR1. Но, в целом, все прозрачно и логично. И, как правило, портирование кода с одного МК на другой, даже если он написан на ассемблере, не составляет большого труда. Особенно если он правильно написан.

Как узнать что есть в конкретном микроконтроллере?
Для этого на каждый МК есть даташит — техническая документация. И вот там, прям на первой странице, написано что почем и как. Вот тебе пример, даташит на Мегу16 с моим закадровым переводом :) Жирным шрифтом помечены опции которые я гляжу в первую очередь, как наиболее интересные для меня, остальное, как правило, присутствует по дефолту.

Features (фичи!)
• High-performance, Low-power AVR® 8-bit Microcontroller
(понтовая экономичная архитектура AVR)

• Advanced RISC Architecture
(просто офигенная вещь для рисковых чуваков!)

– 131 Powerful Instructions – Most Single-clock Cycle Execution
(131 крутая и быстрая команда!)

– 32 x 8 General Purpose Working Registers
(32 восьми разрядных регистра — те самые R0…R31)

– Fully Static Operation
(Полностью статические операции, т.е. тактовая частота может быть хоть 1 импульс в год)

– Up to 16 MIPS Throughput at 16 MHz
(скорость выполнения до 16миллионов операций в секунду!)

– On-chip 2-cycle Multiplier
(а числа умеем множить за два такта! Это правда круто, народ!)

• High Endurance Non-volatile Memory segments
– 16K Bytes of In-System Self-programmable Flash program memory
(памяти хватит набыдлокодить на 16кб кода)

– 512 Bytes EEPROM 8-bit
(и нажрать на века 512 байт мусора в ЕЕПРОМ)

– 1K Byte Internal SRAM
(оперативки 1кб, кому там 2Гигабайт не хватает? Программировать не умеете! =) Тут и 64 байтов за глаза хватает. Помните Билла Гейтса и его «640кб хватит всем!» он знал о чем говорил :)

– Write/Erase Cycles: 10,000 Flash/100,000 EEPROM Microcontroller
(перешивать флеш можно 10тыщь раз, еепром 100тыщь раз. Так что можешь не бояться экспериментировать)

– Data retention: 20 years at 85°C/100 years at 25°C(1)
(Если законсервируешь свой будильник на AVR, то твоих правнуков он еще и через 100 лет порадует)

– Optional Boot Code Section with Independent Lock Bits
In-System Programming by On-chip Boot Program
True Read-While-Write Operation
(поддержка бутлоадеров. Удобная вещь, позволяет прошиваться без программаторов)

– Programming Lock for Software Security In-System
(если жадный и умный, то можешь закрыть от посторонних прошивку и фиг кто выкрадет твои секреты)

• JTAG (IEEE std. 1149.1 Compliant) Interface
– Boundary-scan Capabilities According to the JTAG Standard Programmable
– Extensive On-chip Debug Support
– Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface Flash
(Отладочный интерфейс JTAG и его фичи)

• Peripheral Features
(А вот, собственно и периферия пошла)

– Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
(два таймера 8ми разрядных, с кучей всяких режимов разных.

– One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode ATmega16
(один 16ти разрядный таймер счетчик, с кучей всяких примочек и фишек)

– Real Time Counter with Separate Oscillator
(таймер может тикать от отдельного генератора, удобно если хочешь сделать часы)

– Four PWM Channels ATmega16L
(Четыре ШИМ канала — на тех же таймерах)

– 8-channel, 10-bit ADC
(восьмиканальный 10ти разрядный АЦП. Фичи его ниже)

8 Single-ended Channels
(можно замерять по очереди сразу 8 разных напряжений)

7 Differential Channels in TQFP Package Only
(7 дифференциальных каналов. Правда только в корпусе TQFP т.к. ног у него больше)

2 Differential Channels with Programmable Gain at 1x, 10x, or 200x
(два дифференциальных канала с программируемым усилением)

– Byte-oriented Two-wire Serial Interface
(Поддержка IIC с аппаратным кодированием байтов)

– Programmable Serial USART
(Последовательный интерфейс. Удобен для связи с компом)

– Master/Slave SPI Serial Interface
(SPI интерфейс, пригодится)

– Programmable Watchdog Timer with Separate On-chip Oscillator
(Спец таймер защиты от зависаний)

– On-chip Analog Comparator
(Тот самый компаратор)

• Special Microcontroller Features
(полезные свистоперделки)

– Power-on Reset and Programmable Brown-out Detection
(защита от косяков в работе при пониженном напряжении ака севшие батарейки)

– Internal Calibrated RC Oscillator
(А еще можно сэкономить 20рублей на покупке внешнего кварца. Он нафиг не нужен! :) И это круто!)

– External and Internal Interrupt Sources
(Есть внешние прерывания. Очень удобная вещь)

– Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby
and Extended Standby
(Дофига режимов энергосбережения)

• I/O and Packages
– 32 Programmable I/O Lines
– 40-pin PDIP, 44-lead TQFP, and 44-pad QFN/MLF
(число полезных ножек, тем самых вводов выводов)

• Operating Voltages designs.
– 2.7 — 5.5V for ATmega16L
– 4.5 — 5.5V for ATmega16
(Питающие напряжения. Помните я говорил про низковольтные серии — вот они, во всей красе)

• Speed Grades
– 0 — 8 MHz for ATmega16L
– 0 — 16 MHz for ATmega16
(А это максимальные частоты для разных серий. Низковольтные лажают. Впрочем, они подвержены разгону)

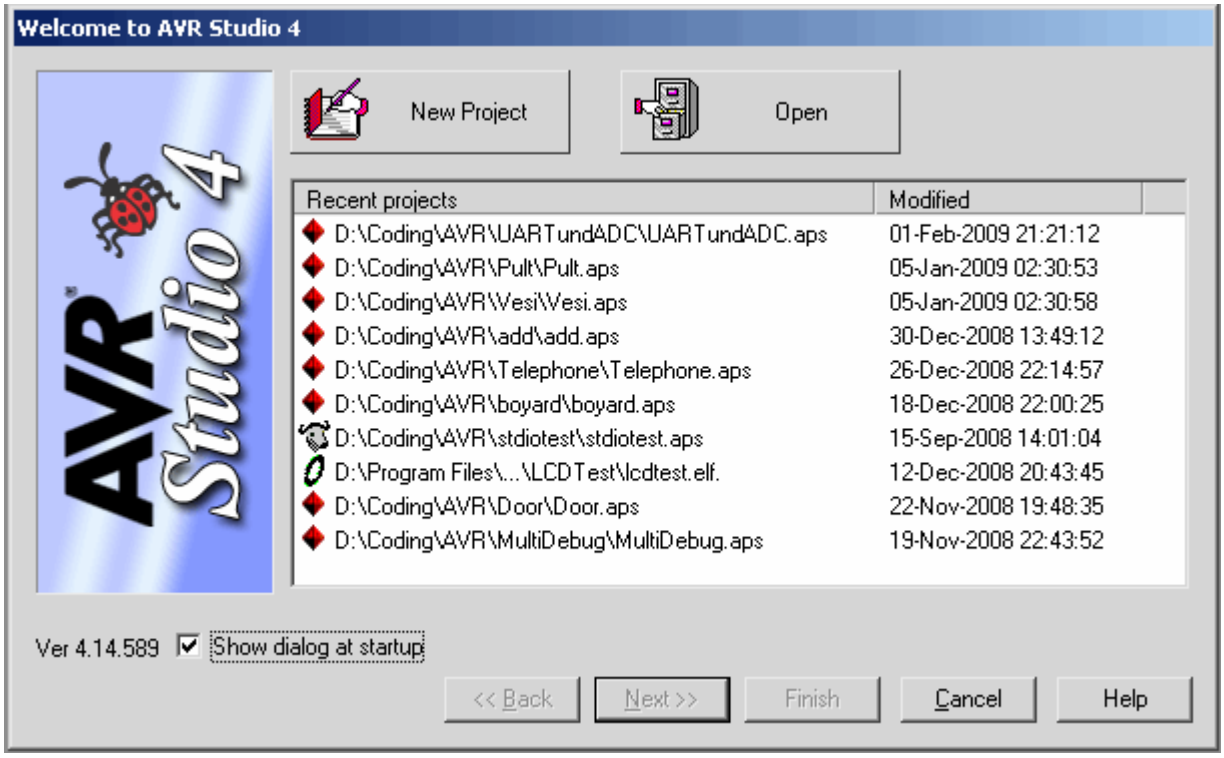
• Power Consumption @ 1 MHz, 3V, and 25°C for ATmega16L
– Active: 1.1 mA
– Idle Mode: 0.35 mA
– Power-down Mode: < 1 µA
(Потребляемая мощность в разных режимах. 1мА даже в активном режиме это фигня. В 10 раз меньше самого тухлого светодиода)

AVR Studio ликбез

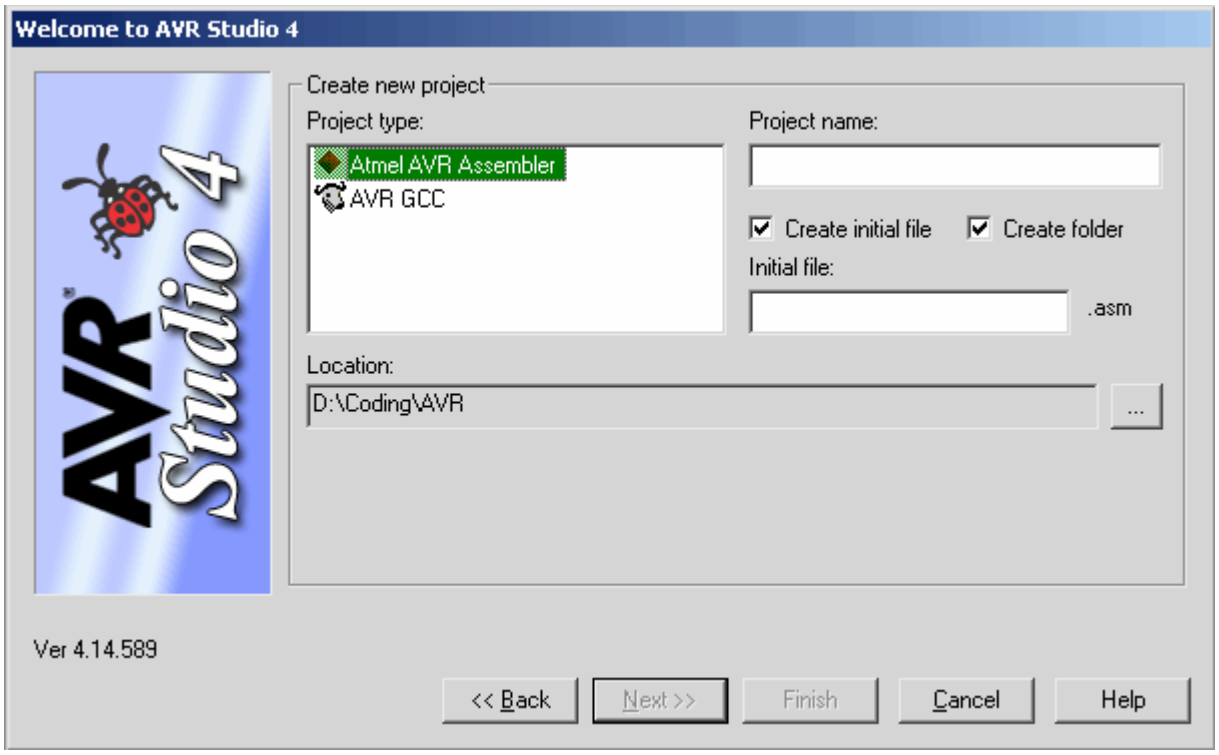
Установка.
Сначала надо с сайта **atmel.com** [1] скачать последнюю версию **AVR Studio**. Весит она что то около 30 метров. Можно и старые релизы использовать, не преступно, но там может не оказаться новых микроконтроллеров. **AVR Studio**, как и многие буржуйские программы, крайне хреново понимает русские имена и длинные пути. Поэтому ставь ее по максимально простому пути, что то вроде **C:\AVR** А сами проекты тоже держи как можно ближе к корню, У меня, например, это **D:\Work\AVR** — никаких имен длинней 8 символов и, конечно же, никаких русских символов. Привет родимый DOS, как говорится.

Да, если планируешь (если не планируешь, то все равно скачай и поставь, не помешает) писать на Си, то рекомендую скачать **GCC** aka **WinAVR** и установить ее ДО студии, туда же, поближе к корню. Тогда студия подхватит ее в качестве своего плагина. Если поставить после, то тоже, может быть подхватит, но возможны проблемы.

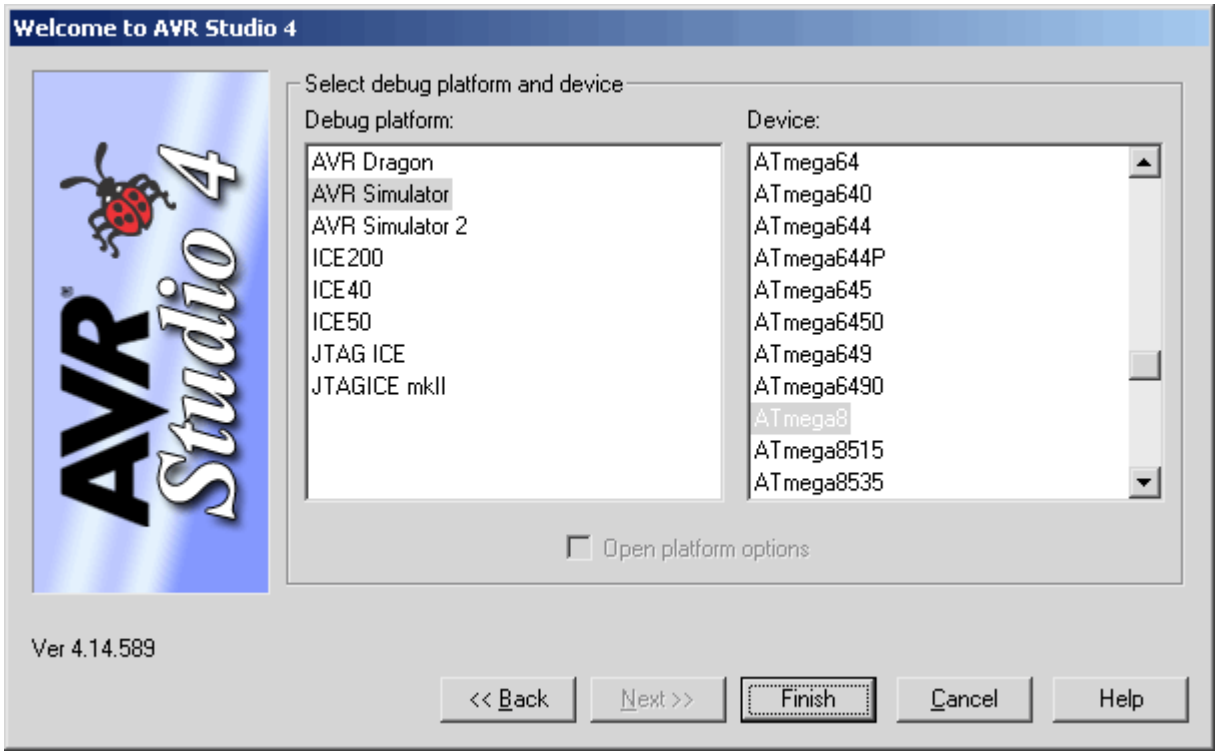
Первый запуск и знакомство с оболочкой
При старте Студия сразу же кидается в тебя мастером создания нового проекта.



В центре будут уже созданные проекты, а нас интересует кнопка **New Project**. Жми ее и вводи параметры будущей разработки.

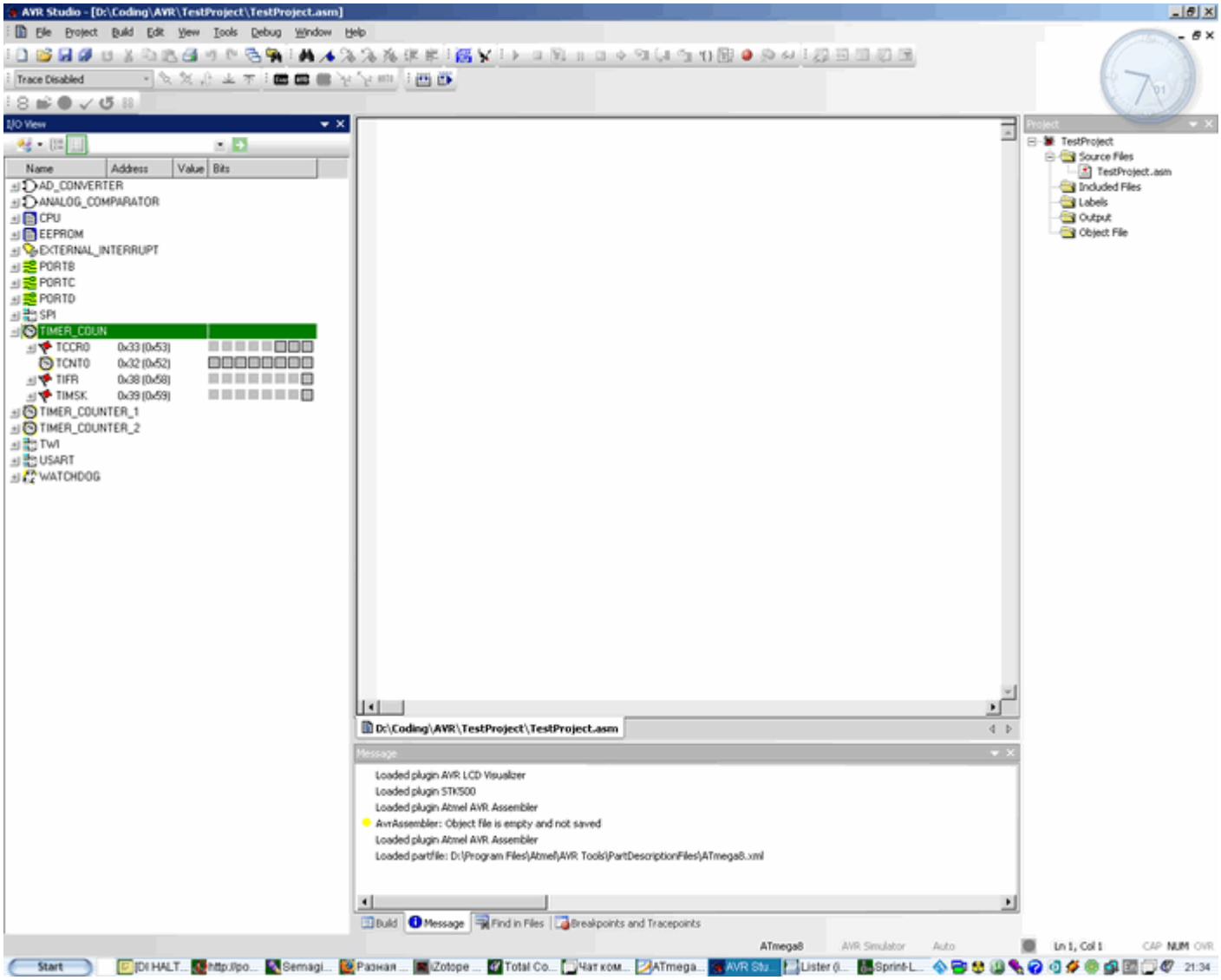


Выбирай тип языка на котором будешь писать: Си или Ассемблер, а также имя будущего проекта. Также не помешает указать путь к папке с проектами. Студия умная и запомнит ее раз и навсегда и под каждый новый проект будет генерить новую папку. Так что скоро там будет полный гадюшник :)))))) Дальше есть две кнопки **Finish** и **Next**. Тебе надо жать на **Next**. Нажмешь на финиш и получишь пустой проект в котором даже процессор то не определен. А вот если **Next**, то будет следующий диалог:



Сильно сомневаюсь, что у тебя первоначально сразу же будет **AVR Dragon** или какой нибудь **ICE 2**, так что мы будем симулировать. Выбирай симулятор. Чем отличается **AVR Simulator** от **AVR Simulator2** я так и не понял, работают одинаково. Выбрал? Вот теперь FINISH HIM! В смысле жми **Finish**.

Во, появилось пустое окно проекта. Оглядим что тут есть:



[Увеличить скриншот](#) [2]

Слева — окно ресурсов МК. Тут ты будешь глядеть что происходит с твоим контроллером. Какие где биты стоят, что на портах, что в счетных регистрах, что в регистрах конфигурации. В центре окно кода, справа же окно проекта — тут будут все файлы показаны. Внизу, под окном кода, отрыг сообщений компилятора. Там все ошибки и косяки отображаются при компиляции проги.

Вверху главное меню. Все там на хоткеях, поэтому заучивай их сразу и будет тебе счастье.

Подробно по пунктикам. Первый ряд.

- Бинокль — поиск в коде. Кроме поиска там есть еще **Mark ALL** — пометить строки с найденым барахлом. Появляются такие голубенькие меточки возле строк.
- Синий флажок и компания — поставить/снять пометку вручную. Удобнейшая вещь эти пометки. Пользуйся!
- Две фиговины для форматирования блоков текста. Типа выделил процедуру, а потом подвинул ее влево или вправо. Сишники оценят :)
- Синяя муть и птичка — кнопки для программаторов понтовых. Если их у тебя нет, то забей.
- Стрелка **Play** — запуск симуляции.

- Квадратик **Stop** — остановка симуляции. Все просто.
- Листок со стрелочкой — запуск программы.
- Пауза — остановка программы там где получилось
- синяя стрелка аля **Enter** — виртуальный **Reset** контроллера
- Желтая стрелка — показать где программа находится сейчас. Жутко полезно когда проект побит на части, и ты пытаешься найти где прога стоит в данный момент.
- Три стандартные кнопки любого отладчика **Step Into** — сделать шаг по тексту программы. **Step Over** — сделать шаг по тексту, но не заходить в процедуры и функции (процедуры и функции при этом выполняются, просто нам это не показывают). **Step-Out** — Выход из процедуры на уровень выше. Конечно, назад МК шагать не умеет, но вот если там цикл какой, то **Step Out** это равносильно тому, что МК сделает одну итерацию цикла и встанет над инструкцией.
- Фигурные скобки и стрелка — выполнить программу до текущего положения курсора. Пощелкай этой кнопочкой и сам поймешь что это
- Красная точка — **BreakPoint** поставить бряк. О брейпоинтах ниже.
- Крестик и красная жопа — нажмешь ее и всем брейкпоинтам в проге настанет жопа — их удалит.
- Очки — это **Watch**, способ подглядывать за какой нибудь переменной, адресом в памяти или регистром. О них тоже потом.
- Кнопки переключалок окон, включать выключать разные блоки интерфейса. Потыкайся по ним и поймешь сам.

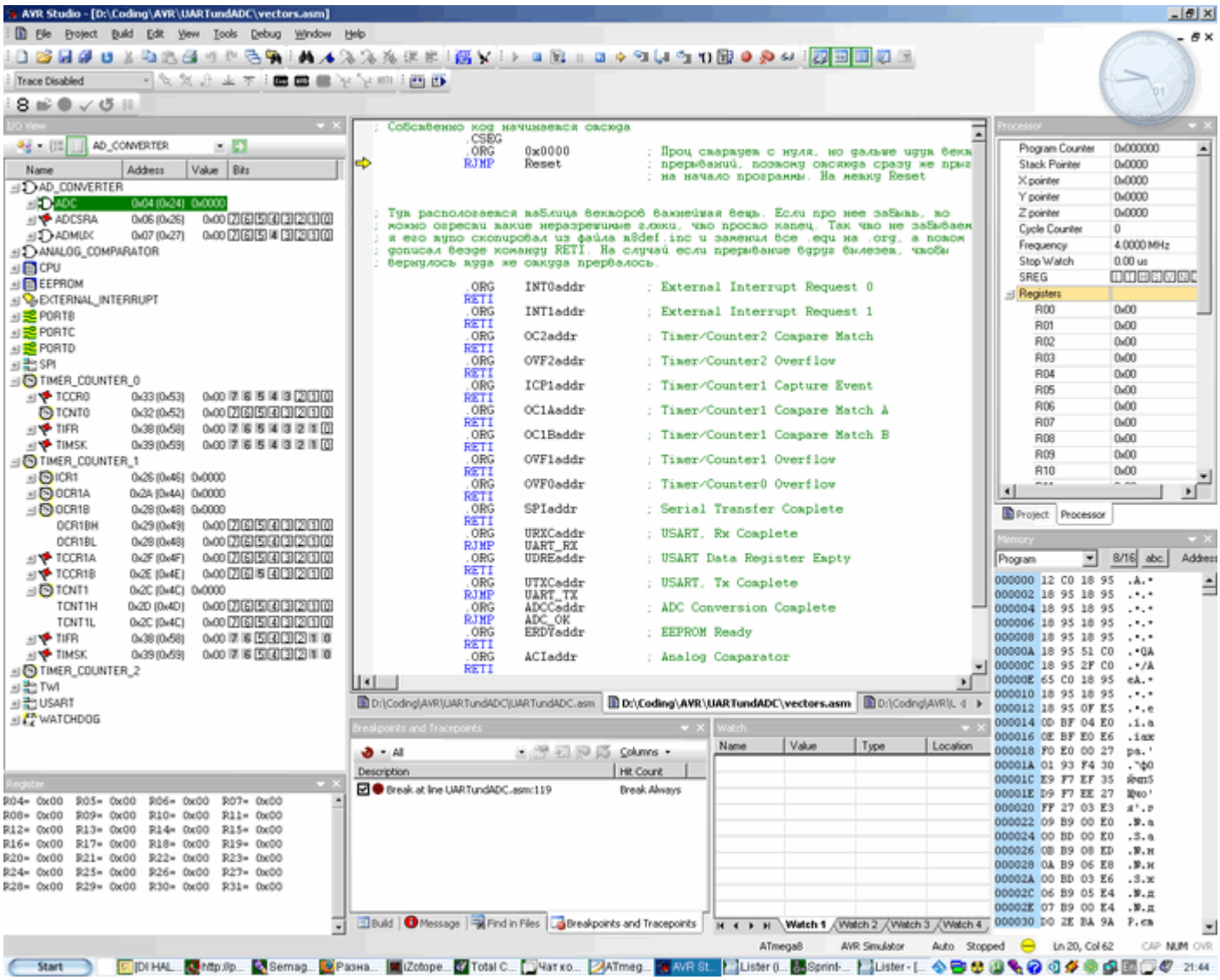
Второй ряд:
Тут ничего не работает, т.к. нет подключенных девайсов (JTAG или STK500) и нас интересуют только две последние кнопки.
Первая из них это компиляция проекта, вторая компиляция и запуск симуляции.

Теперь если ты забьешь в проект простейший код, например,

```
1               .include "m8def.inc"   ; Используем ATMega8
2               NOP
```

Куда уж проще :) То сможешь скомпилировать проект и запустить процесс отладки.

При запуске симулятора повылазит куча разных новых окошек, о которых я тебе сейчас расскажу. Гляди на скриншот:



[Увеличить скриншот](#) [3]

I/O View
Окно переферии, то что справа. Обрати внимание на заголовок окна I/O View — там есть строка поиска и ряд кнопочек. Потыкай их, чтобы получить наиболее удобное представление. Я обычно ставлю **Tree View**. Каждый регистр можно развернуть. Каждый бит подписан и во всплывающей подсказке написано его назначение. Также настоятельно рекомендую прошерстить контекстное меню в этом блоке и выставить там галку **Show Bit numbers** — удобно. Черный квадратик — бит есть. Белый — бита нет. Там же указано значение в байте. В процессе отладки каждый бит можно вручную принудительно выставить или сбросить. Сэмулировав, например, нажатие кнопки.

Processor
Слева есть окно Processor, в нем указана вся подноготная ядра в текущий момент.

- Program Counter — счетчик адресов. По нему можно узнать куда уйдет программа на следующем шаге. Когда отлаживаешь всякие индексные переходы, да извращенские конечные автоматы с переходами вида:

```
1         LDI R16,AddresL ; Кто сказал, что индексные переходы только по спец
2 регистрам?
3         LDI R17,AddresH
4         PUSH R17
5         PUSH R16        ; ICALL? Щаз! Откуда хотим оттуда и скачем.
          RET             ; Наш извращенский метод!
```

- Вот когда такие конструкции отлаживаешь PC становится твоим любимым регистром :)
- Туда же Stack Pointer — указатель стека. Сначала глядим туда, потом в память куда он указывает, и палим где у нас косяк со стеком. Либо если SP не вышел на 0, а по идее доложен был, то у нас срыв стека и неизбежная упячка пыщь пыщь!!!111.
- Слово из трех букв — три индексные пары X, Y, Z — вообщет это те же регистры от 26 до 31го, а тут для удобства выведены.
- Cycle Counter — просто счетчик тиков процессора. Ни разу не пригождался.
- Frequensy — текущая частота эмулируемого процессора. Сразу выставляй ее правильно. Как ее сменить? О! Золотой вопрос. Тому кто придумал прятать эту настройку в задницу надо гвоздь в голову забить. Ничего, нашли, читай ниже.
- Stop Watch — а вот это мега вещь! Время от старта. По нему можно вычислять сколько времени выполняется тот или иной кусок кода. Например задержка. Только изначально он показывается в микросекундах. Нам такие величины обычно не нужны, поэтому лезь в контекстное меню и переключай его в миллисекунды.
- SREG это понятно — флаги.
- Ну и 32 регистра. Правда под регистры я выделил отдельное окошко (слева внизу). Кстати, можно вручную во время эмуляции вписать любое значение в регистр.

Memory
Окно памяти. Можно в списке выбирать любое адресное пространство. Хочешь флеш, хочешь епром, или ОЗУ.

Watch
Это Watch лист. Можно на любой адрес, переменную или любой регистр навесить гляделку и там всегда будет отображаться его содержимое. Удобно при отладке, чтобы не шерстить по коду. Также содержимое регистров/переменных подсвечивается в подсказке при наведении мыши.

Отладка в AVR Studio
Код ты написал, он у тебя даже скомпилировался. Но от радости прыгать не получается — не работает как задумано. Чтож, бывает. Начинаем избавлять код от лажи — отлаживать.
Сначала подготовь плацдарм и выстави требуемую частоту процессора.

Как выставить частоту процессора в AVR Studio:
Загадка века, между прочим. Сходу фиг найдешь. Короче, запускай процесс симуляции. Только в этом случае появится нужный пункт меню. А потом лезь в меню **Debug -> AVR Simulations Options**. Ну не козлы, а? В такие гребеня прятать столь важный параметр. Я почти два месяца его в свое время искал. Там же можно выставить адрес бутлоадера (он зависит от Fuse Bits в реальном МК).

Итак, симуляция запущена, а желтая стрелка бодро указывает на первую команду. Потыкай на клавишу F11, погляди как процессор шагает. Если все нормально, то переходи на то место, где у тебя предпологаемый затык. Например, не выставляется бит в порт. Можно поставить в это место курсор и нажать Run to Cursor, это если по быстрому. Но лучше использовать брякпоинт.

BreakPoint
Это точка останова. Т.е. если какое то условие совпадет, то процессор встанет как вкопанный, пока ты не примешь решение, что же делать дальше. Мощнейшее средство отладки. Самый простой случай это установка Breakpoint — кнопкой F9, она же его и убирает. Через контекстное меню брейпоинт можно временно дезактивировать. Все, если теперь ты нажмешь на F5 то проц, пойдет молотить код подряд, пока не дойдет до брейкпоинта. Где встанет как вкопаный, перейдя в пошаговый режим.

Техническое отступление, при желании можно его пропустить
Не получилось? Прождал полтора часа, а бряк так и не наступил? Ну значит где то у тебя косяк, зациклило прогу. Жми паузу и смотри где переклинило процессор. Можешь пошагово потрейсить и посмотреть в каком именно месте оно крутится. А дальше уже думать. Отработка больших циклов, может быть очень длительной. Например, задержка длительностью в двадцать секунд, эмулируется в Студии порядка пяти минут!!! (на моем древнем Athlon 950) так что если у тебя где то тупит, то не помешает глянуть на показания StopWatch — может на самом деле все еще нормально, просто подождать надо. Чтобы не тупить на таких циклах я их на время отладки закомменчиваю или меняю предделитель таймера с 1024, на 1. На логику же это не влияет, так что можно проскочить их по быстрому.

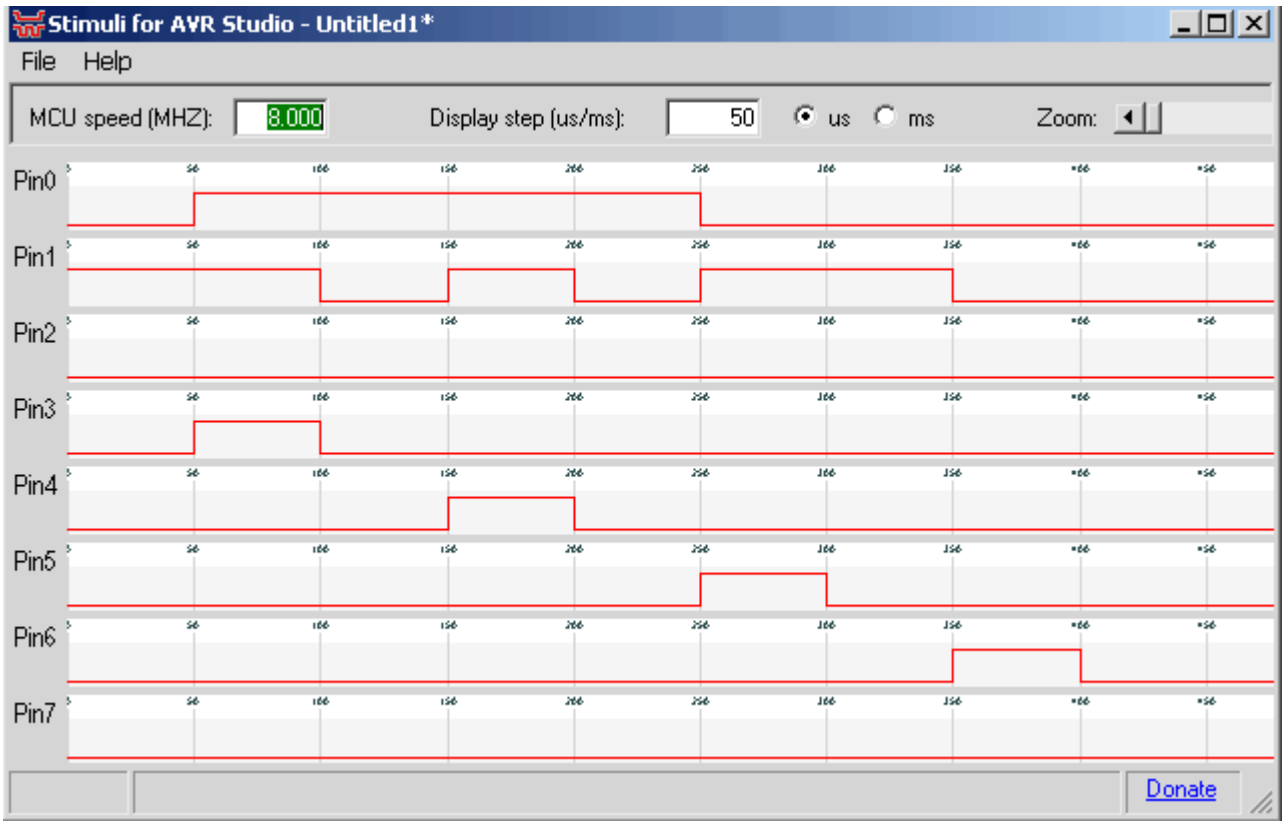
Бряки бывают разные. Добавляются Брейкпоинты из меню **Debug -> New Breakpoint** их там два вида. **Programm Breakpoint** — это тупо точки останова на конкретном месте. Их проще расставлять не отсюда, а прям в коде на нужной строке. А вот **;Data Breakpoint** это уже куда интересней. Погляди на выпадающий список — есть где развернуться. Тут тебе и совпадение битов, и равенство нужному числу, и просто обращение к

адресу/регистру/памяти. Причем можно указывать сразу битовую маску. А в разделе **Location** выбрать любой байт памяти или регистр конфигурации. Мало того, в настройках брейка можно выбрать после скольких событий должен сработать этот бряк. Например, ты поставил брейк на начало цикла, нужно тебе поглядеть что происходит на какой-нибудь 140 итерации. Не будешь же ты тыкать пока оно там все 140 раз не прокрутится. Ставим в свойствах бряка число хитов которые он должен пропустить — 140 и на 140 итерации он тормознет программу. Удобно, черт возьми!
Все брейкпоинты видны в окне **Breakpoint and tracepoint**. Которое возникает внизу, там же где и сообщения об ошибках, в виде закладки. Оттуда им можно менять свойства, там же можно вывести отображение числа хитов и другие свойства бряка.

Работа с портами, эмуляция кнопок и прочего внешнего оборудования.
Есть в отладке **AVR Studio** одно небольшое **западло**, точнее особенность. Когда ты устанавливаешь **порт на вход**, делая подтяжку на резистор:
DDR= 0
PORT=1
То вывод виртуального МК остается равен нулю!!! Таким образом, все кнопки в отладчике AVR Studio по дефолту оказываются нажатыми! Нужно вручную выставить значение PIN, протыкав соответствующие галочки. Неудобно, но так.
А еще можно заказать вывод лога из порта или заливку туда данных извне! Вот это ваще мега фича. Во время отладки, в меню Debug->Simulation Options в разделе Stimuli&Loggin можно выбрать на какой порт, повесить либо логгер, либо же загнать дамп нужных циферок. Входные данные задаются в файле *.sti в формате такого вида [номер таката]:[значение] примерно так:

000000006:01
000000010:02
000000014:03
000000018:04
000000022:05
000000026:06
000000030:07

Можно задавать его вручную, можно написать программу в той же студии, которая тебе его сгенерит :)))) А можно применить одну зашибенную программку **stimuligenerator_net20** [4], которую написал широко известный в узких кругах товарищь ARV обитающий на сайте arv.radioliga.com [5] Кстати, рекомендую к посещению, достойный ресурс.
Программка проста как мычание — выставляешь нужные биты по времени, не забыв указать частоту принимающего МК (чтобы такты выставило правильно). Опа и никаких забот :) Для работы программка правда требует .NET окружение, поэтому вначале придется скачать 30МБ .Net Frame Work если ты его себе еще не поставил.



Выходящий лог можно наблюдать в Студии в окне Message (вкладка рядом с сообщениями об ошибках компиляции) и потом поглядеть в файле. Формат там такой же (номер такта:значение порта). Такты идут по порядку при старте процессора, с нуля. Его же можно также открыть stimuligenerator [4]'ом. Короче, программа Must Have.

Работа с прерываниями
Ну тут все просто, хочешь прерывание — ткни вручную бит ее флага, мол вот оно, и прерывание начнется. Особенно это актуально для всяких внешних источников. Вроде UART или INT. После ручного вызова, например, RX прерывания не грех будет взять и вбить ручками в нужный регистр якобы принятое значение. Опа и словно так и было :)))))

Ладно, на сегодня хватит. Потом еще парочку телег по отладке в AVR Studio толкну. Мне, например, уже давно никакие эмуляторы вроде Proteus не нужны. На все хватает родимой студии, да UART вывода. Ну еще парочка примочек, но о них в следующий раз.

AVR Studio в Linux.

Печально, но факт, что Atmel штампует свое замечательное IDE только для семейства OS Windows. Поэтому пользователям Linux приходится по-всякому извращаться, чтобы заполучить себе на машину этот удобнейший инструмент разработчика. Существует два очевидных решения этой задачи. Первый — запускать студию в Windows, которая крутится на виртуальной машине (отдельную машину с Win не рассматриваем). Второй — запускать программу посредством Wine.

Первый способ хорош 100%-й совместимостью и полным отсутствием софтверных граблей. Железные же практически полностью висят на разработчиках виртуальных машин и обычно стремятся к нулю. Однако, виртуальная машина отжирает у хост-машины ценные системные ресурсы. Кроме того, стоит учесть, что налог на Windows не зависит от того насколько виртуальна машина, на которой бегает эта операционная система. Ну и окно-в-окне, особенно при неудачно реализованном механизме переключения между системами, удовольствия не доставляет. Если вас эти ограничения не смущают, это вполне себе вариант. В Википедии есть прекрасная сводная таблица виртуальных машин [1], которая может вам пригодиться.

Способ запуска программ (любых) посредством Wine — вполне себе самодостаточен. Стоит помнить, что проект постоянно развивается. Сегодняшний Wine не чета тому, который мне довелось пощупать при первом знакомстве. Сайт проекта: http://www.winehq.org/ [2]. Этот вариант распишу подробно.

Итак. Для начала обновляем Wine. Это обязательно нужно сделать, т.к. в состав дистрибутивов обычно входит стабильный, а значит достаточно старый релиз. Последний *стабильный* релиз — 1.0.1 (октябрь 2008 года). У меня сейчас стоит версия 1.1.33, на момент написания статьи отрелизилась 1.1.36 (8 января 2010 года). Идем в раздел Downloads на сайте, **ищем СВОЙ дистрибутив** и внимательно читаем инструкции по установке. В моем случае это Debian, о нем и пойдет речь дальше. Если у вас не Debian — ставите по-своему.

Подключаем репозиторий Вайна. Нужно добавить строчку в файл /etc/apt/sources.list (с правами root'а):

```
#echo "deb http://www.lamaresh.net/apt lenny main" >> /etc/apt/sources.list
```

Скачаем ключик для доступа к репозиторию:

```
$wget http://www.lamaresh.net/apt/key.gpg
#apt-key add key.gpg
```

Обновим список пакетов и установим программу:

```
#apt-get update
#apt-get install wine
```

После установки последуем инструкции [3] от aor_dreamer с форума AVR Freaks.

Качаем и запускаем скрипт winetricks:

```
$wget http://www.kegel.com/wine/winetricks
$bash winetricks
```

В открывшемся окне ставим следующие флажки:

```
- corefonts
- dcom98
- gdiplus
- gecko
- mdac28
- msxml3
- vcrun2005
- allfonts
- fakeie6
```

После того как скрипт отработает (придется принять участие и посоглашаться с лицензиями MS) качаем AVR Studio [4] с оффсайта Atmel и устанавливаем:

```
$wine AvrStudio4Setup.exe
```

Запускаем (это путь, куда студия становится по умолчанию):

```
$wine "~/.wine/drive_c/Program Files/Atmel/AVR Tools/AvrStudio4/AVRStudio.exe"
```

По желанию можно написать простенький скрипт для запуска и повесить его на рабочий стол (правильную картинку можно взять здесь [5]).

Осталась небольшая проблемка. AVR Studio отказалась видеть подключенный аналог STK500 (модифицированный HVProg). Интерфейс с ПК в моей версии реализован посредством FT232RL [6], которая видна в системе как /dev/ttyUSB0. Студия же ищет программатор по привычным COMx, где x — номер порта.

Тут все просто. Создаем ссылку (root'ом):

```
#ln -s /dev/ttyUSB0 <home_dir>/.wine/dosdevices/com1
```

/dev/ttyUSB0 — физическое устройство (например, реальный COM порт: /dev/ttyS0);
< home_dir> — путь к домашнему каталогу пользователя;
com1 — имя порта для Win (например, lpt1 для параллельного интерфейса).

Программатор определился в автоматическом режиме и тут же бодро прочитал/зашил пару МК. Пока проверял только ISP режим, но думаю с HV проблем не будет. (также наверняка не будет и проблем с JTAG ICE первой модификции, а также всеми программаторами работающими по честному COM порту — прим.DI HALT)

Учтите, что пользователь, которому нужен доступ к портам через wine, должен иметь права на чтение-запись в порт, т.е. находиться в соответствующей группе.

Блог автора: http://devmind.livejournal.com/ [7]

AVR. Учебный курс. Макроассемблер

Перед изучением системы команд микроконтроллера надо бы разобраться в инструментарии. Плох тот плотник который не знает свой топор. Основным инструментом у нас будет компилятор. У компилятора есть свой язык — макроассемблер, с помощью которого жизнь программиста упрощается в разы. Ведь гораздо проще писать и оперировать в голове командами типа MOV Counter,Default_Count вместо MOV R17,R16 и помнить что у нас R17 значит Counter, а R16 это Default_Count. Все подстановки с человеческого языка на машинный, а также многое другое делается средствами препроцессора компилятора. Его мы сейчас и рассмотрим.

Комментарии в тексте программы начинаются либо знаком «**;**«, либо двойными слешами «**//**«, а еще AVR Studio поддерживает Сишную нотацию комментариев, где коменты ограничены «колючей проволокой» **/* коммент */**.

Оператор .include позволяет подключать в тело твоей программы кусок кода из другого текстового файла. Что позволяет разбить большую исходник на кучу мелких, чтобы не загромождать и не мотать туда сюда огромную портянку кода. Считай куда ты воткнул **.include** туда и вставился кусок кода из другого файла. Если надо подключать не весь файл, а только его часть, то тебе поможет директива **.exit**дойдя до которой компилятор выйдет из файла.

Оператор .def позволяет привязать к любому слову любое значение из ресурсов контроллера — порт или регистр. Например сделал я счетчик, а считаемое значение находится в регистре **R0**, а в качестве регистра-помойки для промежуточных данных я заюзал **R16**. Чтобы не запутаться и помнить, что в каком регистре у меня задумано я присваиваю им через **.def** символические имена.

```
1 .def    schetchik = R0
2 .def    pomoika = R16
```

И теперь в коде могу смело использовать вместо официального имени **R0** неофицальную кличку **schetchik** Одному и тому же регистру можно давать кучу имен одновременно и на все он будет честно откликаться.

Также есть оператор **.undef** после которого компилятор напрочь забывает, что данной переменной что либо соответствовало. Иногда бывает удобно. Когда одно и то же символическое имя хочется присвоить разным ресурсам.

```
1 .undef  pomoika
```

Оператор .equ это присвоение выражения или константы какой либо символической метке.
Например, у меня есть константа которая часто используется. Можно, конечно, каждый раз писать ее в коде, но вдруг окажется, что константа выбрана неверно, а значит придется весь код шерстить и везде править, а если где-нибудь забудешь, то получишь такую махровую багу, что задолбаешься потом ее вылавливать. Так что нафиг, все константы писать надо через
.equ! Кроме того, можно же присвоить не константу, а целое выражение. Которое при компиляции посчитается препроцессором, а в код пойдет уже исходное значение. Надо только учитывать, что деление тут исключительно целочисленное. С отбрасыванием дробной части, без какого-либо округления, а значит 1/2 = 0, а 5/2 = 2

```
1 .equ    Time = 5
2 .equ    Accelerate = 4
3 .equ    Half_Speed = (Accelerate*Time)/2
```

Директивы сегментации. Как я уже рассказывал в посте про архитектуру контроллера AVR [1] память контроллера разбита на независимые сегменты — данные (**ОЗУ**), код (**FLASH**), **EEPROM**

Чтобы указать компилятору, что где находится применяют директивы сегментации и адресации.

.CSEG сегмент кода, он же флеш. После этой директивы идет тело программы, комманды процессора. Тут же можно засунуть какие нибудь данные которые не меняются, например таблицу с заранее посчитаными значениями, статичный текст или таблицу символов для знакогенератора.

В сегменте кода уместны директивы:
Адресная метка. Любое слово, не содержащее пробелов и не начинающееся с цифры, главное, чтобы после него стояло **двоеточие**.

```
1 .CSEG
2 label:   LDI      R16,'A'
3          RJMP   label
```

В итоге, после компиляции вместо label в код подставится адрес команды перед которой стоит эта самая метка, в данном случае адрес команды LDI R16,'A'
Адресными метками можно адресовать не только код, но и данные, записанные в любом сегменте памяти. Об этом чуть ниже.

.ORG address означет примерно следующее «копать отсюда и до обеда», т.е. до конца памяти. Данный оператор указывает **с какого адреса пойдет собственно программа**. Обычно используется для создания таблицы прерываний.

```
1          .CSEG
2          .ORG 0x0000
3          RJMP Start  ;перепрыгиваем таблицу векторов.
4
5          .ORG INT0addr  ; External Interrupt0 Vector Address
6          RJMP INT0_expection
7
8          .ORG INT1addr  ; External Interrupt1 Vector Address
9          RETI
10
11         .ORG OC2addr   ; Output Compare2 Interrupt Vector Address
12         RJMP PWM_1
13
14         .ORG OVF2addr  ; Overflow2 Interrupt Vector Address
15         RETI
16
17         .ORG ICP1addr  ;Input Capture1 Interrupt Vector Address
18         RETI
19
20         .ORG 0x0032            ; Начало основной программы
21
22 Start:  LDI R16,0x54   ; и понеслась
```

Статичные данные пихаются в флеш посредством операторов

.dbмассив байтов.
.dw массив слов — два байта.
.dd массив двойных слов — четыре байта
.dq массив четверных слов — восем байт.

```
1 Constant:       .db     10      ; или 0xAh в шестнадцатеричном коде
2 Message:        .db     "Привет лунатикам"
3 Words:          .dw     10, 11, 12
```

В итоге, во флеше вначале будет лежать число 0A, затем побайтно будут хекскоды символов фразы «привет лунатикам», а дальше **000A, 000B, 000C**.
Последнии числа, хоть сами и невелики, но занимают по два байта каждое, так как обьявлены как **.dw**.

.DSEG сегмент данных, оперативка. Те самые жалкие считанные байты. Сюда не зазорно пихать перменные, делать тут буффера, тут же находится стек.

Тут действует оператор **.BYTE** позволяющий указать на расположение данных в памяти.

```
1 var1:           .BYTE 1
2 table:          .BYTE 10
```

В первом случае мы указали переменную **var1** состоящую из одного байта.
Во втором случае у нас есть цепочка из 10 байт и переменная **table** указывающая **на первый байт из цепочки**.
Адрес остальных вычисляется смещением.
Указывать размеры перменных нужно для того, чтобы компилятор их правильно адресовал и они не налезали друг на друга.

.EESEG сегмент EEPROM, энергонезависимая память. Можно писать, можно считывать, а при пропаже питания данные не повреждаются.
Тут действуют те же директивы что и в **flash — db, dw, dd, dq**.

MACRO — оператор макроподстановки. Вот уж реально чумовая вещь. Позволяет **присваивать имена целым кускам кода**, мало того, еще параметры задавать можно.

```
1 .MACRO SUBI16          ; Start macro definition
2         subi @1,low(@0)        ; Subtract low byte
3         sbci @2,high(@0)       ; Subtract high byte
4 .ENDM                          ; End macro definition
```

@0, @1, @2 это параметры макроса, они нумеруются тупо по порядку. А при вызове **подставляются в код**.

Вызов выглядит как обычная команда:

```
1 SUBI16  0x1234,r16,r17
```

После имени через запятую передаются параметры, которые подставятся в код.
Макросы позволяют насоздавать себе **удобных команд** на все случаи жизни, по сути создать свой язык. Но надо помнить, что каждый макрос это тупо кусок кода, поэтому если макрос получается большой, то его лучше оформить в виде процедуры или функции — будет резкая экономия места в памяти, но выполняться будет чуток медленней.

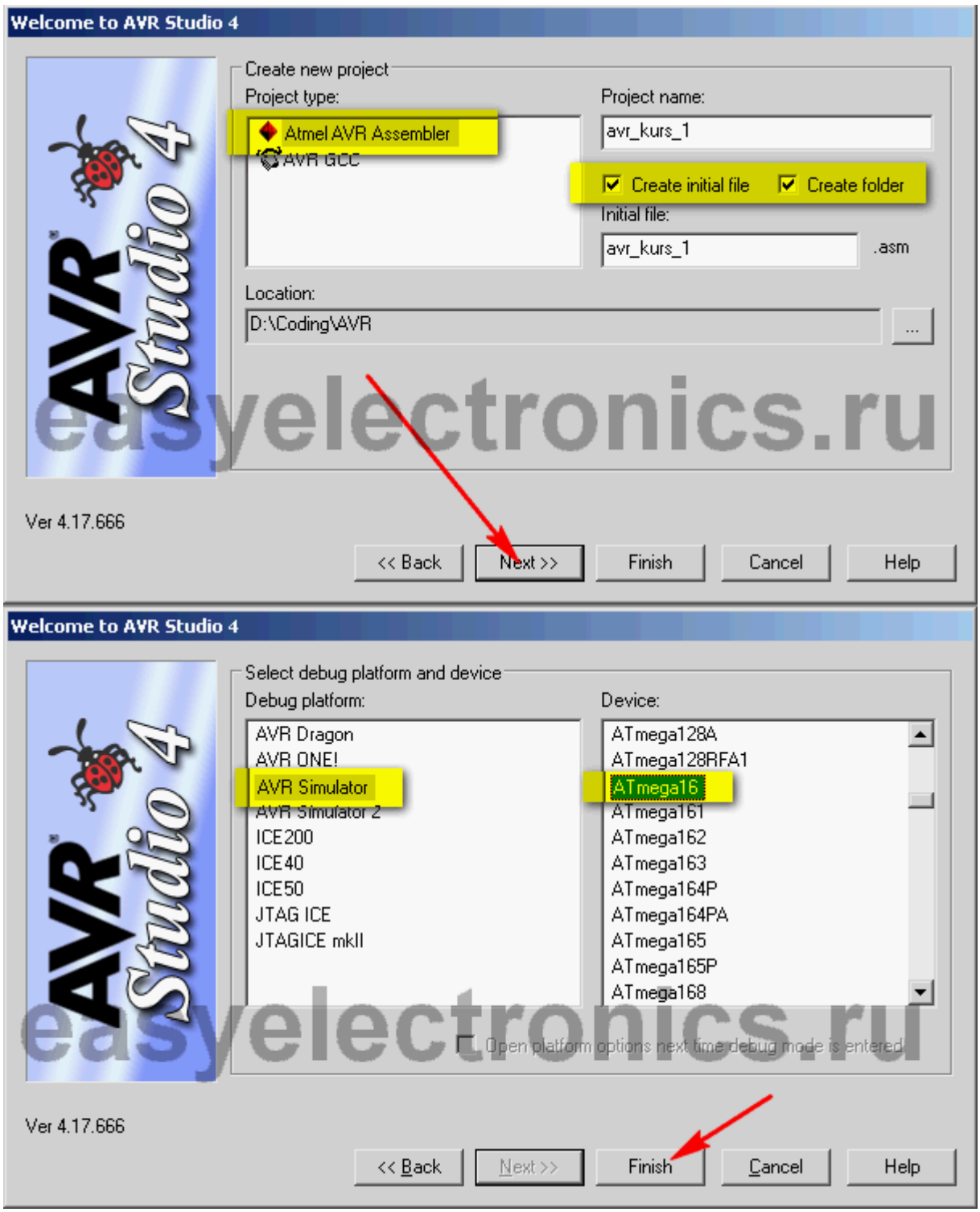
Макроассемблер это мощнейшая штука. По ходу пьесы я буду вводить разные макросы и показывать примеры работы макроопределений.

AVR. Учебный курс. Простейшая программа.

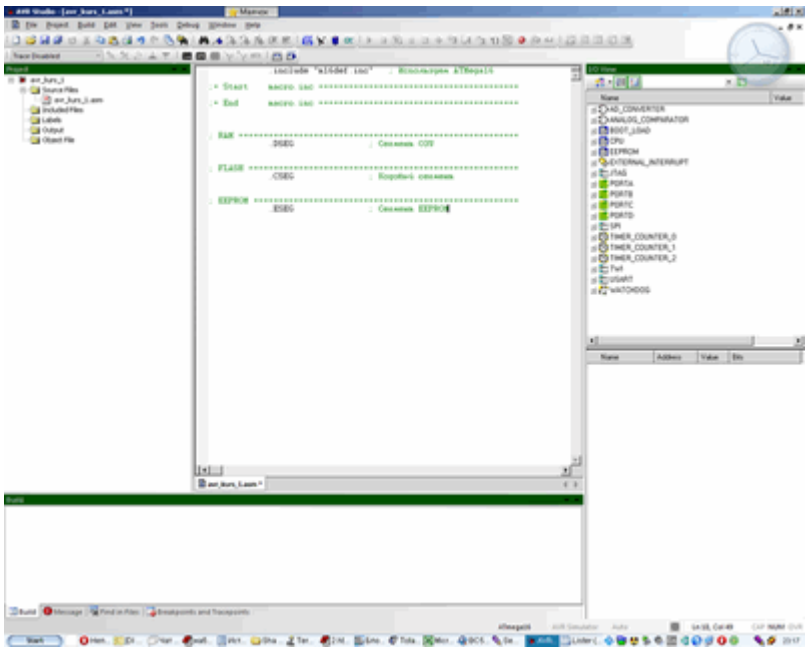
Итак, прежде чем что то делать надо понять как вообще выполняется программа в контроллере, как работает ядро процессора. Для этого нам хватит AVR Studio и его эмулятора. Не очень интересно, может даже занудно, но если этот шаг пропустить, то дальнейшие действия будут как бег в темноте.

Поскольку в демоплате Pinboard [1] используется процессор ATmega16, то рассматривать мы будем именно его. Впрочем, как я уже говорил, для других контроллеров AVR это также будет справедливо. Отличия, конечно, есть, но они не существенные.

Запускаем AVR Studio (далее просто студия) и в выскочившем мастере сразу же создаем проект:



Откроется окно:



Увеличить [2]

Оно может выглядеть чуток не так. Дело в том, что интерфейс студии очень легко конфигурируется и перекраивается под себя. Так что можно перетащить панели как нам удобно. Что я и сделал.

Систему команд хорошо бы распечатать себе на листочке. Их там всего около 130, кратким списком (тип команды, что делает и какие операнды) занимает пару листов формата А4. Учить не надо, прочитать раз на десять, чтобы помнить знать что у нас есть. Даже я периодически подглядываю в систему команд, хотя пишу на ассемблере уже много лет.
Я же команды которые буду использовать буду описывать по мере появления.

В центральном окне пишем код:

```
1                .include "m16def.inc"    ; Используем ATMega16
2  ;= Start macro.inc ======================================
3
4  ; Тут будут наши макросы, потом.
5
6  ;= End macro.inc  ========================================
7
8
9  ; RAM =================================================
10               .DSEG                    ; Сегмент ОЗУ
11
12
13 ; FLASH ===============================================
14               .CSEG                    ; Кодовый сегмент
15
16
17 ; EEPROM ==============================================
18               .ESEG                    ; Сегмент EEPROM
```

Это вроде шаблона для начала любого проекта. Правда кода тут 0 байт :) Только директивы.
Обрати также внимание на оформление кода. Дело в том, что в ассемблере нет никаких структурных элементов — весь код плоский, тупо в столбик. Так что поэтому чтобы он был читабелен его надо в обязательно порядке разбивать на блоки, активно пользоваться табуляцией и расставлять комментарии. Иначе уже через пару дней ты не сможешь понять, что вообще написал.

В коде будут активно использоваться макросы. Поначалу они будут писаться в основном файле, в секции macro.inc, но потом я их вынесу в отдельный файл, чтобы не мешались. Так удобней.

Как я уже говорил, в микроконтроллере куча раздельных видов памяти. И тут используется гарвардская архитектура. Т.е. исполняемый код в одном месте, а переменные в другом. Причем это разные адресные пространства. Т.е. адрес 0000 в сегменте данных это не то же что и адрес 0000 в сегменте кода. Это дает большое преимущество в том, что данные не могут испортить код, но не дает писать полиморфные программы. Такая архитектура является традиционной для микроконтроллеров.

У обычного PC компа используется другая архитектура — Фон Неймановская. Там данные и код находятся в одном адресном пространстве. Т.е., скажем, с адреса 0000 по 0100 идет код, а с 100 до FFFF данные.

В нашей же программе и код и данные располагаются на одном листе, но чтобы компилятор понял где у нас что они выделяются директивами сегментации. Пока нас интересуют директива .CSEG после нее начинается исполняемый код. Там и будем писать нашу программу.

Возьмем и напишем:

```
1 ; FLASH ===================================================
2                 .CSEG           ; Кодовый сегмент
3                 NOP
4                 NOP
5                 NOP
```

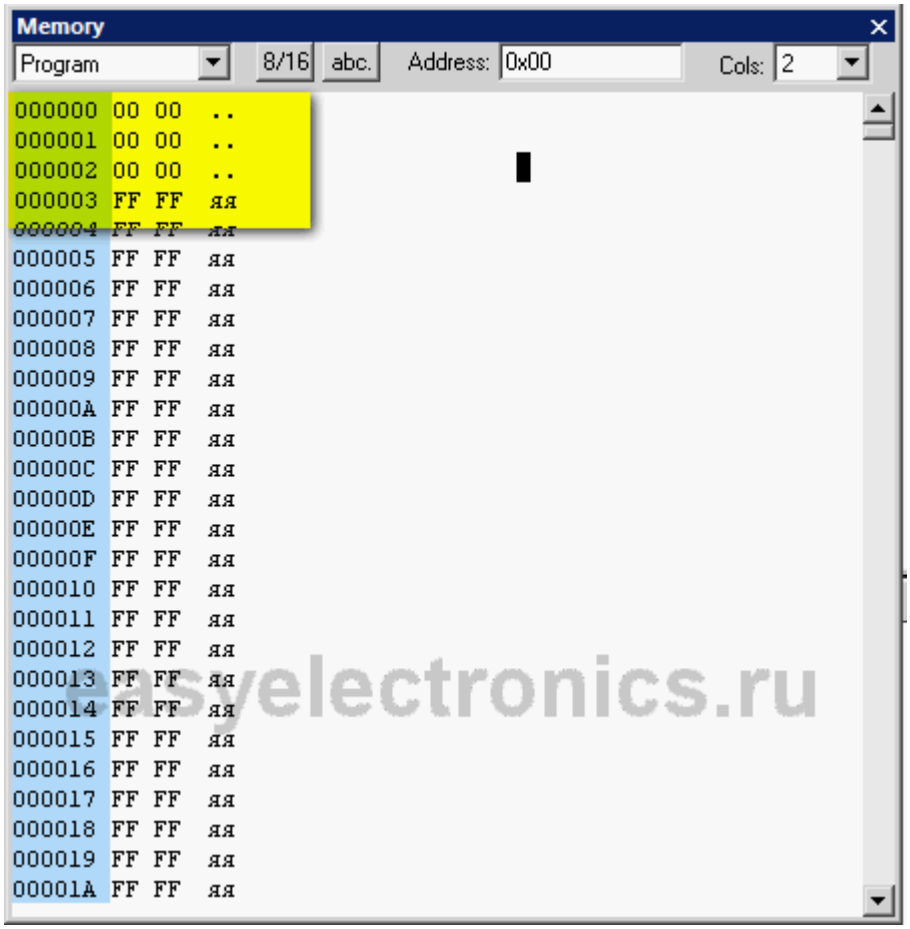
Что мы сделали? А ничего! Команда NOP это команда затычка. Она не делает ничего, просто занимает 2 байта и 1 такт.

Речь не о команде NOP (там и обсуждать то нечего), а о том как оно все будет выполняться.

Запускай симуляцию (Ctrl+F7) когда пробежит прогресс бар компиляции/симуляции и возле первого NOP возникнет стрелочка нажми ALT+O — выскочит диалог настройки симуляции. Там тебе надо там только выставить частоту 8Мгц. Почему 8Мгц? Просто на Pinboard [1] частота главного проца по дефолту 8Мгц. Если у тебя свои идеи на этот счет — можешь поправить как угодно. На симуляцию это не влияет.

Вернемся к нашей симуляции. Давай посмотрим как выглядит наша программа с точки зрения машинных кодов, как размещается в памяти. Интерес, по большей части, чисто теоретический, редко когда пригождается и по этому во многих учебных курсах по AVR на это даже внимание не заостряют. А зря! Т.к. упускается глубинное ощущение кода. Открой окно просмотра программ. View — Memory или Alt+4.

Там выбери тип памяти Programm. Это и есть наш Flash сегмент. Выставим в списке cols две колонки, чтобы было наглядней.

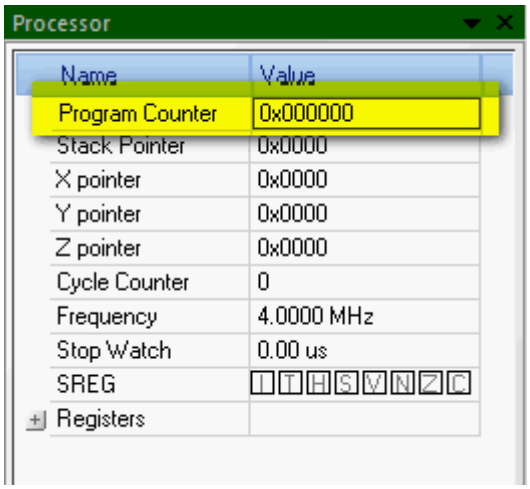


Это наша память программ. На голубом фоне идут адреса, они, как видишь, в словах. Т.е. двум байтам соответствует один адрес. Но, на самом деле это интерпретация адресации компилятором, микроконтроллер же может оперировать в памяти программ с точностью до байта.

0000 это код команды NOP. У нас три команды, поэтому шесть нулей.

Обрати внимание на то, что команды идут с адреса 0000. Это потому, что мы не указали ничего иного. А кодовый сегмент начинается с адреса 0000.

Теперь смотрим на программный счетчик **Programm Counter**, он показывает адрес инструкции которая будет выполнена.



Поскольку мы только стартанули, то он равен нулю — нулевая инструкция. Нажми F11, процессор сделает шаг, программный счетчик изменится на 1 и покажет следующую инструкцию. И так до тех пор, пока не дойдет до

третьего NOP, а что у нас после него? А после него у нас FF до конца памяти. FF это несуществующая инструкция, на ней виртуальный контроллер перезагрузится с ошибкой invalid opcode, а реальный контроллер ее проигнорирует, пробежав по ним, как по NOP, до конца памяти.

Сбрось симуляцию микроконтроллера (Shift+F5) и вручную выстави в Program Counter адрес 0×000002 — проц сам перепрыгнет на последнюю команду NOP. Если менять Program Counter то проц будет выполнять те команды, которые мы ему укажем. Но как это сделать в реальном контроллере? В него то мышкой не залезешь!

Программный счетчик меняется командами переходов. Их много (условные, безусловные, относительные), о них я расскажу подробней чуть позже, пока же приведу один пример.

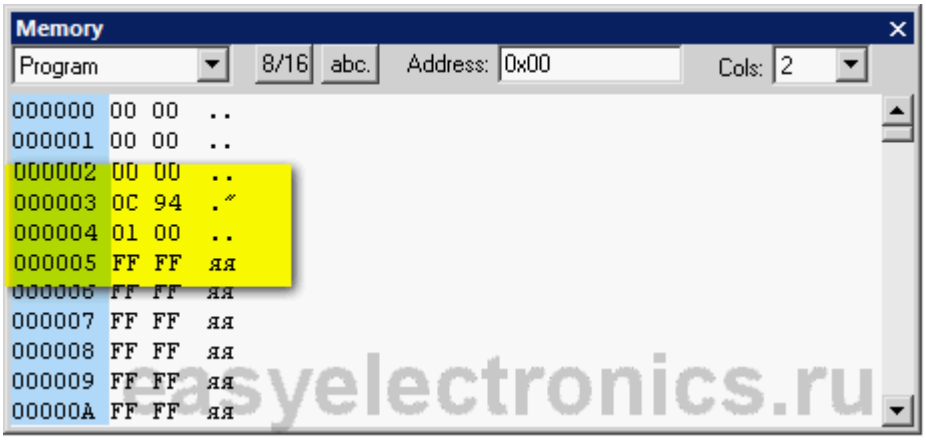
Добавим в наш код команду JMP и ее аргумент — адрес 0×000001.

```
1                .CSEG                    ; Кодовый сегмент
2                NOP
3                NOP
4                NOP
5                JMP 0x000001
```

Команда JMP, как и все команды перехода, работает просто — записывает в Program Counter свой аргумент. В нашем случае — 0×000001.

Перекомпиль проект и посмотри на то, как меняется Program Counter (далее буду звать его PC) и вообще как теперь идет процесс выполнения программы. Видишь, после JMP в программный счетчик заносится новый адрес и процессор сразу же перепрыгивает в начало кода, но не на первую, а на вторую инструкцию. Программа зациклилась.

Это называется абсолютный переход. Он может сигануть куда угодно, в любую область памяти программ. Но за такую дальнобойность приходится платить. Если заглянешь в память, то увидишь там такую картину:



0C 94 — это код нашей комады, а 01 00 адрес перехода. Длина команды стала четыре байта. Два байта ушло на адрес. Вот, кстати, особенность памяти в том, что там данные записываются зеркально, т.е. младший байт числа по младшему адресу (порядок little-endian).

Но поскольку дальнобойные команды применяются редко, основной командой перехода в AVR является относительный переход RJMP. Основное отличие тут в том, что в PC не записывается не точный адрес куда надо перейти, а просто к PC прибавляется смещение. Вот так:

```
1                .CSEG                          ; Кодовый сегмент
2                NOP
3                NOP
4                NOP
5                RJMP PC+2
6                NOP
7                NOP
8                RJMP PC-6
```

```
9                NOP
```

По идее, должно работать напрямую, т.е. RJMP +2. Но компилятор такую запись не понимает, он оперирует абсолютными адресами, приводя их потом в нужные смещения машинного кода. Поэтому применим макроопределение PC — это текущее значение счетчика в данном месте программы. Т.е.

```
1 JMP PC
```

Наглухо зациклит программу в этой строчке.

Благодаря относительному переходу, смещение можно запихать в те же два байта, что занимает команда. При этом код относительного перехода выглядит так Cx xx, где xxx это смещение от +/-2047 команд. Что обычно хватает с лихвой. В памяти же команда перехода выглядит как xx Cx, то есть байты переставлены.

Таким образом, длина команды у относительного перехода составляет 2 байта, против четырех у абсолютного. Экономия!!! Учитывая что в обычной программе подобные переходы на каждом шагу.

Вот только что же, каждый раз самому вручную высчитывать длину перехода? Ладно тут программка в три команды, а если их сотни? Да если дописал чуток то все переходы заново высчитывать?

Угу. Когда то давно, когда компиляторов еще не было так и делали. Сейчас же все куда проще — компилятор все сделает сам. Ему только надо указать откуда и куда. Делается это с помощью меток.

```
1                .CSEG           ; Кодовый сегмент
2 M1:            NOP
3                NOP
4                NOP
5                RJMP M2
6                NOP
7 M2:            NOP
8                RJMP M1
9                NOP
```

Метки могут быть из букв или цифр, без пробелов и не начинаться с цифр. Заканчиваются двоеточием. По факту, метка означает текущий адрес в словах. Так что ее можно использовать и в операциях. Надо только помнить, что она двубайтная, а наш контроллер однобайтный. Разобрать двубайтное значение по байтам можно с помощью директивы компилятора Low и High

Например,

```
1                .CSEG                  ; Кодовый сегмент
2  M1:           NOP
3                NOP
4                LDI     ZL,low(M2)     ; Загрузили в индекс
5                LDI     ZH,High(M2)
6
7                IJMP
8
9                NOP
10               NOP
11               NOP
12 M2:           NOP
13               RJMP M1
14               NOP
```

Команда LDI загружает непосредственное значение в регистр старшей (от R16 до R31) группы. Например, LDI R17,3 и в R17 будет число 3. А тут мы в регистры R30 (ZL) загрузили младший байт адреса на который указывает метка M2, а в R31 (ZH) старший байт адреса. Если протрассируешь выполнение (F11) этого кода, то увидишь как меняются значения регистров R30 и R31 (впрочем 31 может и не поменяться, т.к. там был ноль, а мы туда ноль и

запишем — адрес то мал). Смену значений регистров можно поглядеть в том же окне где и Program Counter в разделе Registers.

Команда IJMP это косвенный переход. Т.е. он переходит не по адресу который заложен в коде операции или идет после него, а по адресу который лежит в индексной регистровой паре Z. Помните я говорил, что шесть последних регистров старшей регистровой группы образуют три регистровые пары X,Y,Z используются для адресации? Вот это я и имел ввиду.

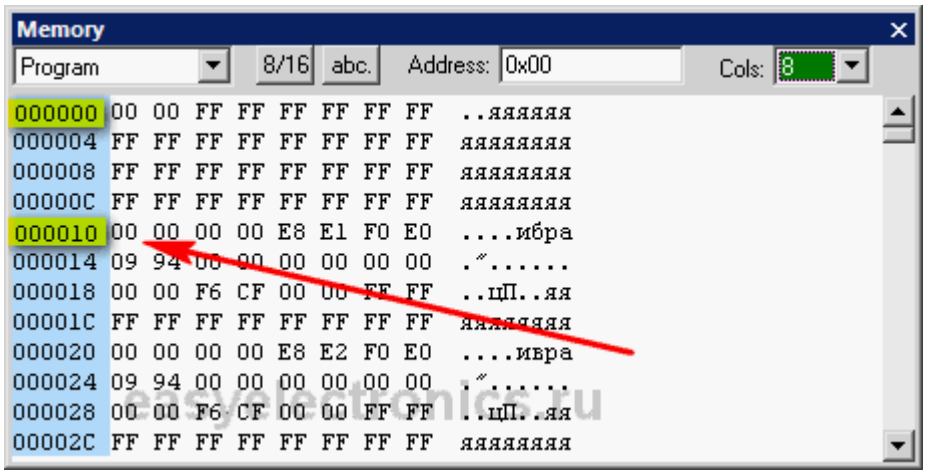
После IJMP мы переходим на нашу же M2, но хитровывернутым способом. Зачем вообще так? Не проще ли применить RJMP и JMP. В этом случае да, проще.

Но вот благодаря косвенному переходу мы можем **программно** менять точку перехода. И это просто зверский метод при обработке всяких таблиц и создании switch-case структур или конечных автоматов. Примеры будут позже, пока попробуйте придумать сами.

Давай подвинем нашу кодовую конструкцию в памяти на десяток байт. Добавь в код директиву ORG

```
1  ; FLASH ===================================================
2                   .CSEG           ; Кодовый сегмент
3                   NOP
4
5                   .ORG 0x0010
6  M1:              NOP
7                   NOP
8                   LDI     ZL,low(M2)      ; Загрузили в индекс
9                   LDI     ZH,High(M2)
10
11                  IJMP
12
13                  NOP
14                  NOP
15                  NOP
16 M2:              NOP
17                  RJMP M1
18                  NOP
```

Скомпиль и запусти. Открой память и увидишь что в точке с адресом 0000 у нас стоит наш NOP (Точка входа должна быть, иначе симулятор скукожит от такого когнитивного диссонанса. А вот дальше сплошные FF FF и лишь начиная с нашего адреса 0×0010 пошли коды остальных команд.



То есть мы разбили код, заставив директивой ORG компилятор перетащить его дальше, начиная с 0×0010 адреса.

Зачем это нам потребовалось? Ну, в первую очередь, это требуется чтобы перешагнуть таблицу векторов прерываний. Она находится в первых адресах.

Во-вторых, такая мера нужна чтобы записать код в конец флеша, при написании бутлоадеров (о них тоже дальше в курсе будет).

А еще таким образом можно прятать в коде данные, перемешивая данные с кодом. Тогда дизассемблирование такой программы превратится в ад =) своего рода, обфускация публичной прошивки, сильно затрудняющая реверс инженеринг.

Да, у кристалла есть биты защиты. Выставил которые и все, программу можно считать только спилив крышку кристалла с помощью электронных микроскопов, что очень и очень недешево. Но иногда надо оставить прошивку на виду, но не дать сорцов. И вот тогда превращение прошивки в кашу очень даже поможет =)

Ну или написать ее на Си ;)))))) (Есть вариант шифровки прошивки и дешифровки ее на лету бутлоадером, но это отдельная тема. О ней может быть расскажу)

Еще, там, в студии, же где и программный счетчик, есть ряд интересных параметров.
Во-первых, там отдельно вынесены наши индексные пары X,Y,Z и можно не лазать на дно раздела Registers.
Во-вторых, там есть Cycle counter — он показывает сколько машинных циклов протикал наш контроллер. Один машинный цикл в AVR равен одному такту. Frequency содержит частоту контроллера в данный момент, но мы задаем ее вручную.
А в-третьих, есть Stop Watch который показывает время выполнения. В микросекундах. Но можно переключить и в миллисекунды (пошарь в контекстном меню).

Если тебе кажется, что все слишком просто и я чрезмерно все разжевываю. Хы, не расслабляйся, у меня сложность растет по экспоненте. Скоро пойдет работа на прерываниях, а потом будем писать свою операционную систему :))))))

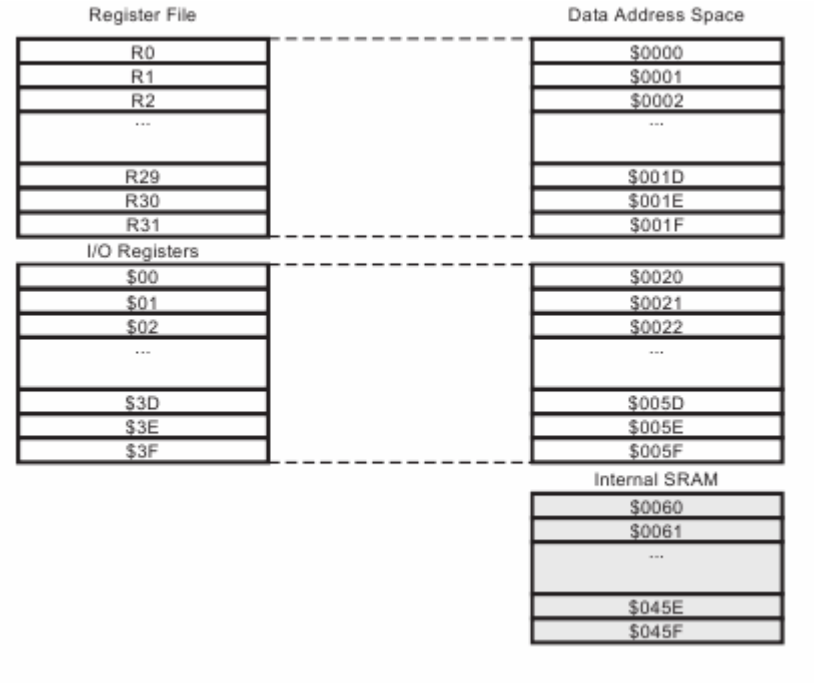
AVR. Учебный Курс. Работа с памятью

Так, с работой ядра на предмет переходов и адресации разобрались. Пора обратить свой взор в другую область — память.

Ее тут два вида (EEPROM не в счет т.к. она вообщет переферия, а о ней потом):

- RAM — оперативка
- ROM — ПЗУ, она же flash, она же память программ

Так как архитектура у нас Гарвардская, то у оперативы своя адресация, а у флеша своя. В даташите можно увидеть структуру адресации ОЗУ.



Сразу обратите внимание на адреса! РОН и регистры периферии, а также ОЗУ находятся в одном адресном пространстве. Т.е. адреса с 0000 по 001F занимают наши регистры, дальше вплоть до адреса 005F идут ячейки ввода-вывода — порты. Через порты происходит конфигурирование всего, что есть на борту контроллера. И только потом, с адреса 0060 идет наше ОЗУ, которое мы можем использовать по назначению.

Причем обратите внимание, что у регистров I/O есть еще своя адресация — адресное пространство регистров ввода-вывода (от 00 до 3F), она указана на левой части рисунка. Блок IO/Register Эта адресация работает ТОЛЬКО в командах OUT и IN Из этого вытекает интересная особенность.

К регистрам периферии можно обратиться двумя разными способами:

- Через команды IN/OUT по короткому адресу в пространстве адресов ввода-вывода
- Через группу команд LOAD/STORE по полному адресу в пространстве адресов RAM

Пример. Возьмем входной регистр асинхронного приемопередатчика UDR он имеет адрес 0x0C(0x2C) в скобках указан адрес в общем адресном пространстве.

```
1         LDI R18,10      ; Загрузили в регистр R18 число 10. Просто так
2
3         OUT UDR,R18     ; Вывели первым способом, компилятор сам
4                         ; Подставит вместо UDR значение 0x0C
5
6         STS 0x2C,R18    ; Вывели вторым способом. Через команду Store
7                         ; Указав адрес напрямую.
```

Оба метода дают идентичные результаты. НО! Те что работают адресацией в пространстве ввода-вывода (OUT/IN) на два байта короче. Это и понятно — им не нужно хранить двубайтный адрес произвольной ячейки памяти, а короткий адрес пространства ввода—вывода влезает и в двухбайтный код команды.

Правда тут возникает еще один прикол. Дело в том, что с каждым годом появляются все новые и новые камни от AVR и мяса в них все больше и больше. А каждой шкварке нужно свои периферийные регистры ввода-вывода. И вот, дожили, в ATMega88 (что пришла на замену Mega8) периферии уже столько, что ее регистры ввода-вывода уже не умещаются в лимит адресного пространства 3F.

Опаньки, приплыли. И вот тут у тех кто пересаживается с старых камней на новые начинаются недоуменные выражения — с чего это команды OUT/IN на одних периферийных регистрах работают, а на других нет?

А все просто — разрядности не хватило.

А ядро то единое, его уже не переделать. И вот тут ATMELовцы поступили хитро — они ввели так называемые memory mapped регистры. Т.е. все те регистры, что не влезли в лимит 3F доступны теперь только одним способом — через Load/Store.

Вот такой прикол. Если открыть какой нибудь m88def.inc то там можно увидеть какие из регистров ввода-вывода «правильные» а какие memory mapped.

Будет там бодяга вот такого вида:

```
1  ; ***** I/O REGISTER DEFINITIONS ****************************************
2  ; NOTE:
3  ; Definitions marked "MEMORY MAPPED"are extended I/O ports
4  ; and cannot be used with IN/OUT instructions
5  .equ   UDR0   = 0xc6   ; MEMORY MAPPED
6  .equ   UBRR0L   = 0xc4   ; MEMORY MAPPED
7  .equ   UBRR0H   = 0xc5   ; MEMORY MAPPED
8  .equ   UCSR0C   = 0xc2   ; MEMORY MAPPED
9  .equ   UCSR0B   = 0xc1   ; MEMORY MAPPED
10 .equ   UCSR0A   = 0xc0   ; MEMORY MAPPED
11
12 бла бла бла, и еще много такого
13
14 .equ   OSCCAL  = 0x66   ; MEMORY MAPPED
15 .equ   PRR     = 0x64   ; MEMORY MAPPED
16 .equ   CLKPR   = 0x61   ; MEMORY MAPPED
17 .equ   WDTCSR  = 0x60   ; MEMORY MAPPED
18 .equ   SREG    = 0x3f  ;<------ А тут пошли обычные
19 .equ   SPL     = 0x3d
20 .equ   SPH     = 0x3e
21 .equ   SPMCSR  = 0x37
22 .equ   MCUCR   = 0x35
23 .equ   MCUSR   = 0x34
24 .equ   SMCR    = 0x33
25 .equ   ACSR    = 0x30
```

Вот такие пироги.

И на этой ниве понимаешь, что в сторону кроссмодельной совместимости ассемблерного кода летит летит большой мохнатый орган с целью наглухо его накрыть. Ведь одно дело подправить всякие макроопределения и дефайны, описывающие регистры, а другое сидеть и, аки Золушка, отделять правильные порты от неправильных.

Впрочем есть решение. Макроязык! Не нравится система команд? Придумай свою с блекджеком и шлюхами! Сварганим свою собственную команду UOUT типо универсальный OUT

```
1         .macro    UOUT
2         .if     @0 < 0x40
3         OUT     @0,@1
4         .else
5         STS     @0,@1
6         .endif
7         .endm
```

Если значение входного параметра @0 меньше 0x40 значит это «правильный» регистр. Если больше — memory_mapped.

Дальше везде, не думая, используем UOUT вместо OUT и не парим себе мозг извратами адресации. Компилятор сам подставит нужную инструкцию.

```
1         UOUT    UDR,R18
```

Аналогично и для команды IN Вообще, такими вот макросами можно ОЧЕНЬ сильно разнообразить ассемблер, превратив его в мощнейший язык программирования, рвущий как тузик тряпку всякие там Си с Паскалями.

Ну так о чем я… а о ОЗУ.

Итак, с адресацией разобрались. Адреса памяти, откуда начинаются пользовательские ячейки ОЗУ теперь ты знаешь где смотреть — в даташите, раздел Memory Map. Но там для справки, чтобы знать.

А в нашем коде оперативка начинается с директивы .DSEG Помните наш шаблончик?

```
1                .include "m16def.inc"   ; Используем ATMega16
2
3  ;= Start macro.inc ================================
4
5  ; Макросы тут
6
7  ;= End  macro.inc ==================================
8
9
10 ; RAM =============================================
11               .DSEG           ; Сегмент ОЗУ
12
13
14 ; FLASH ===========================================
15               .CSEG           ; Кодовый сегмент
16
17
18 ; EEPROM ==========================================
19               .ESEG           ; Сегмент EEPROM
```

Вот после .DSEG можно задавать наши переменные. Причем мы тут имеем просто прорву ячеек — занимай любую. Указал адрес и радуйся. Но зачем же вручную считать адреса? Пусть компилятор тут думает.

Поэтому мы возьмем и зададим меточку

```
1               .DSEG
2 Variables:    .byte   3
3 Variavles2:   .byte   2
```

Директива .byte зарезервирует нам столько байт, сколько мы ей указали. Таким образом, на Variables у нас будет три байта, а на Variables2 два байта.

Если считаем, что у нас Atmega16, а у ней адреса RAM начинаются с 0x0060, то компилятор посчитает адреса так:

Variables = 0×0060
Variables2 = 0×0063

А в памяти это будет лежать следующим образом (приведу в виде линейного списка):

```
1 0x0060 ## ;Variables
2 0x0061 ##
3 0x0062 ##
4 0x0063 ## ;Variables2
5 0x0064 ##
6 0x0065 ## ;Тут могла бы начинаться Variables4
```

В качестве ## любой байт. По дефолту FF. Разумеется ни о какой типизации переменных, начальной инициализации, контроля за переполнениями и прочих буржуазных радостей говорить не приходится. Это Спарта! В смысле, ассемблер. Все ручками.
Если провести аналогию с Си, то это как работа с памятью через одни лишь void указатели. Сишники поймут.

Поймут и ужаснутся. Т.к. мир этот жесток и коварен. Чуть просчитался с индексом — затер другие данные. И хрен ты эту ошибку поймаешь если она сразу не всплывет.

Так что внимание, внимание и еще раз внимание. Все операции с памятью прогоняем через трассировку и ничего у нас не вылезет и не переполнится.

В сегменте данных работает также директива .ORG Работает точно также — переносит адреса, в данном случае меток, от сих и до конца памяти. Одна лишь тонкость — ORG 0000 даст нам самое начало ОЗУ, а это R0 и прочие регистры. А нулевой километр ОЗУ на примере Мега16 даст ORG 0×0060. А в других контроллерах еще какое-нибудь значение. Каждый раз в даташит лазать лениво, поэтому есть такое макроопределение как SRAM_START указывающее на начало ОЗУ для конкретного МК.

Вообще полезно почитать файл m16def.inc на предмет символических имен разных констант.

Так что если хотим начало ОЗУ, скажем 100 байт оставить под какой нибудь мусорный буффер, то делаем такой прикол.

```
1                .DSEG
2                .ORG SRAM_START+100
3
4 Variables:     .byte 3
```

Готово, расчистили себе буфферную зону от начала до 100.

Ладно, с адресацией разобрались. Как работать с ячейками памяти? А для этих целей существует две группы команд. LOAD и STORE самая многочисленная группа команд.

Дело в том, что с ячейкой ОЗУ ничего нельзя сделать кроме как загрузить в нее байт из РОН, или выгрузить из нее байт в РОН.

Записывают в ОЗУ команды Store (ST**), а считываю команды Load (LD**).

Чтение идет в регистр R16…R31, а адрес ячейки задается либо непосредственно в команде. Вот простой пример. Есть трехбайтная переменная Variables, ее надо увеличить на 1. Т.е. сделать операцию Variables++

```
1                .DSEG
2  Variables:    .byte   3
3  Variavles2:   .byte   1
4
5                .CSEG
6
7  ; Переменная лежит в памяти, сначала надо ее достать.
8          LDS     R16, Variables         ; Считать первый байт Variables  в R16
9          LDS     R17, Variables+1       ; Считать второй байт Variables  в R17
10         LDS     R18, Variables+2       ; Ну и третий байт в R18
11
12 ; Теперь прибавим к ней 1, т.к. AVR не умеет складывать с константой, только
13 ; вычитать, приходиться извращаться. Впрочем, особых проблем не доставляет.
14
15         SUBI    R16,(-1)        ; вообще то SUBI это вычитание, но -(- дает +
16         SBCI    R17,(-1)        ; А тут перенос учитывается. Но об этом потом.
17         SBCI    R18,(-1)        ; Математика в ассемблере это отдельная история
18
19         STS     Variables,R16          ; Сохраняем все как было.
20         STS     Variables+1,R17
21         STS     Variables+2,R18
```

А можно применить и другой метод. Косвенную запись через индексный регистр.

```
1                .DSEG
```

```
2  Variables:      .byte   3
3  Variavles2:     .byte   1
4
5                  .CSEG
6  ; Берем адрес нашей переменной
7          LDI     YL,low(Variables)
8          LDI     YH,High(Variables)
9
10 ; Переменная лежит в памяти, сначала надо ее достать.
11         LD      R16, Y+         ; Считать первый байт Variables  в R16
12         LD      R17, Y+         ; Считать второй байт Variables  в R17
13         LD      R18, Y+         ; Ну и третий байт в R18
14
15 ; Теперь прибавим к ней 1, т.к. AVR не умеет складывать с константой, только
16 ; вычитать, приходиться извращаться. Впрочем, особых проблем не доставляет.
17
18         SUBI    R16,(-1)        ; вообще то SUBI это вычитание, но -(- дает +
19         SBCI    R17,(-1)        ; А тут перенос учитывается. Но об этом потом.
20         SBCI    R18,(-1)        ; Математика в ассемблере это отдельная история
21
22         ST      -Y,R18          ; Сохраняем все как было.
23         ST      -Y,R17          ; Но в обратном порядке
24         ST      -Y,R16
```

Тут уже заюзаны операции с постинкрементом и преддекрементом. В первой сначала читаем, потом прибавляем к адресу 1. Во второй вначале вычитаем из адреса 1, а потом сохраняем.

Подобными инкрементальными командами удобно перебирать массивы в памяти или таблицы какие.
А там есть еще и косвенная относительная запись/чтение LDD/STD и еще варианты на все три вида индексов (X,Y,Z). В общем, кури даташит и систему команд.

Стек
О, стек это великая вещь. За что я его люблю, так это за то, что срыв стека превращает работоспособную программу в полную кашу. За то что стековые операции требуют повышенного внимания, за то что если где то стек сорвет и сразу не отследишь, то фиг это потом отловишь… В общем, прелесть, а не штуковина.

Почему люблю? Ну дык, если Си это тупое ремесло, быстро и результативно, то Ассемблер это филигранное искусство. Как маньяки вроде Jim'а из бумаги и только из бумаги [1] клепают шедевры, хотя, казалось бы, купи готовую сборную модель и клей себе в удовольствие. Так и тут — от самого процесса прет нипадецки. В том числе и от затраха с отладкой :))))

Так вот, о стеке. Что это такое? А это область памяти. Работает по принципу стопки. Т.е. какую последнюю положил, ту первой взял.

У стека есть указатель, он показывает на вершину стека. За указатель стека отвечает специальный регистр SP, а точнее это регистровая пара SPL и SPH. Но в микроконтроллерах с малым обьемом ОЗУ, например в Тини2313, есть только SPL

При старте контроллера, обычно, первым делом инициализируют стек, записывая в SP адрес его дна, откуда он будет рости. Обычно это конец ОЗУ, а растет он к началу.

Делается это таким вот образом, в самом начале программы:

```
1          LDI R16,Low(RAMEND)
2          OUT SPL,R16
3
4          LDI R16,High(RAMEND)
5          OUT SPH,R16
```

Где RAMEND это макроопределение указывающий на конец ОЗУ в текущем МК.

Все, стек готов к работе. Данные кладутся в стек командой PUSH Rn, а достаются через POP Rn.
Rn — это любой из РОН.

Еще со стеком работают команды CALL, RCALL, ICALL, RET, RETI и вызов прерывания, но об этом чуть позже.

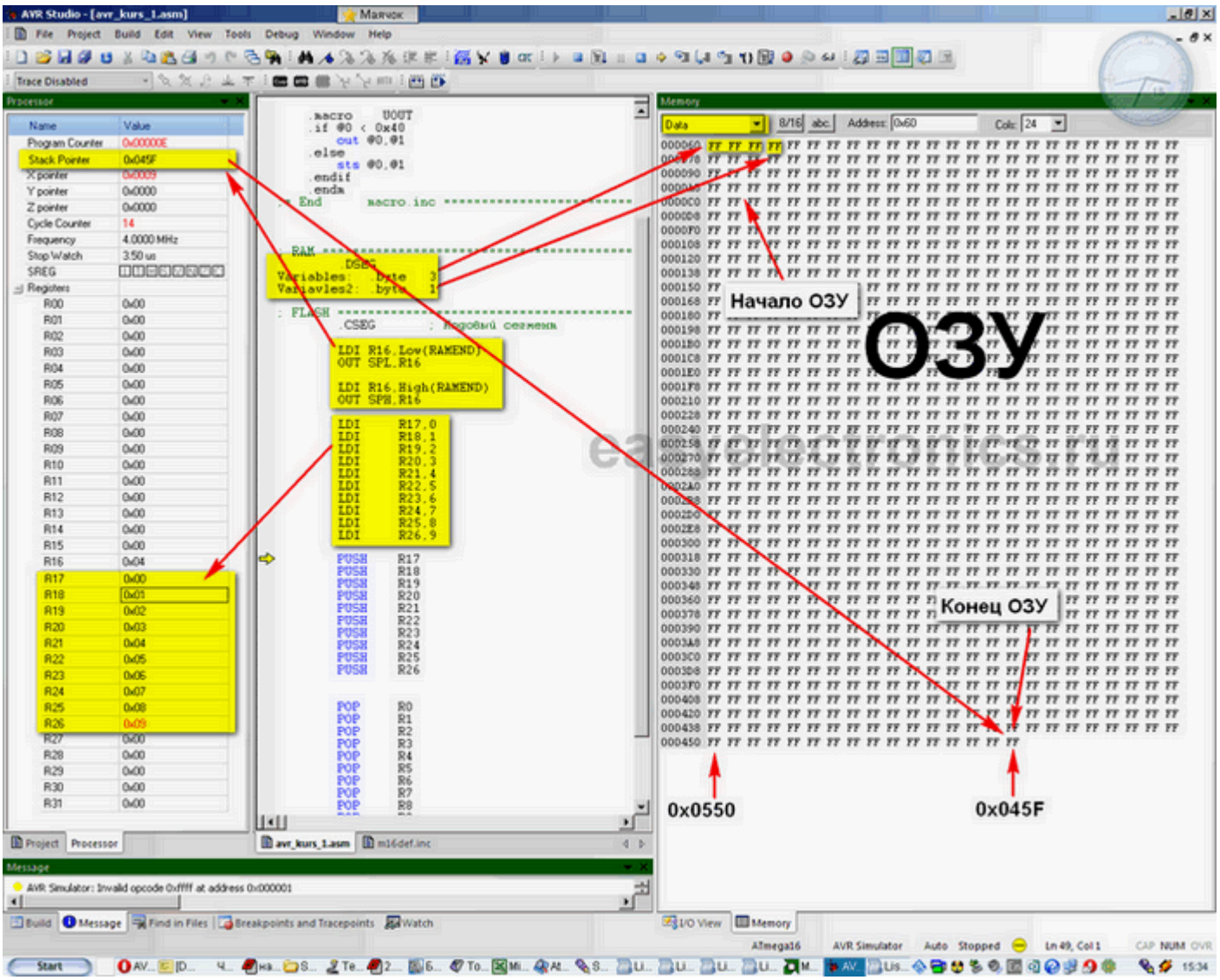
Давай-ка поиграемся со стеком, чтобы почувствовать его работу, понять как и куда он движется.

Вбей в студию такой код:

```
1                .CSEG                  ; Кодовый сегмент
2
3                LDI R16,Low(RAMEND)    ; Инициализация стека
4                OUT SPL,R16
5
6                LDI R16,High(RAMEND)
7                OUT SPH,R16
8
9                LDI     R17,0   ; Загрузка значений
10               LDI     R18,1
11               LDI     R19,2
12               LDI     R20,3
13               LDI     R21,4
14               LDI     R22,5
15               LDI     R23,6
16               LDI     R24,7
17               LDI     R25,8
18               LDI     R26,9
19
20               PUSH    R17             ; Укладываем значения в стек
21               PUSH    R18
22               PUSH    R19
23               PUSH    R20
24               PUSH    R21
25               PUSH    R22
26               PUSH    R23
27               PUSH    R24
28               PUSH    R25
29               PUSH    R26
30
31
32               POP     R0      ; Достаем значения из стека
33               POP     R1
34               POP     R2
35               POP     R3
36               POP     R4
37               POP     R5
38               POP     R6
39               POP     R7
40               POP     R8
41               POP     R9
```

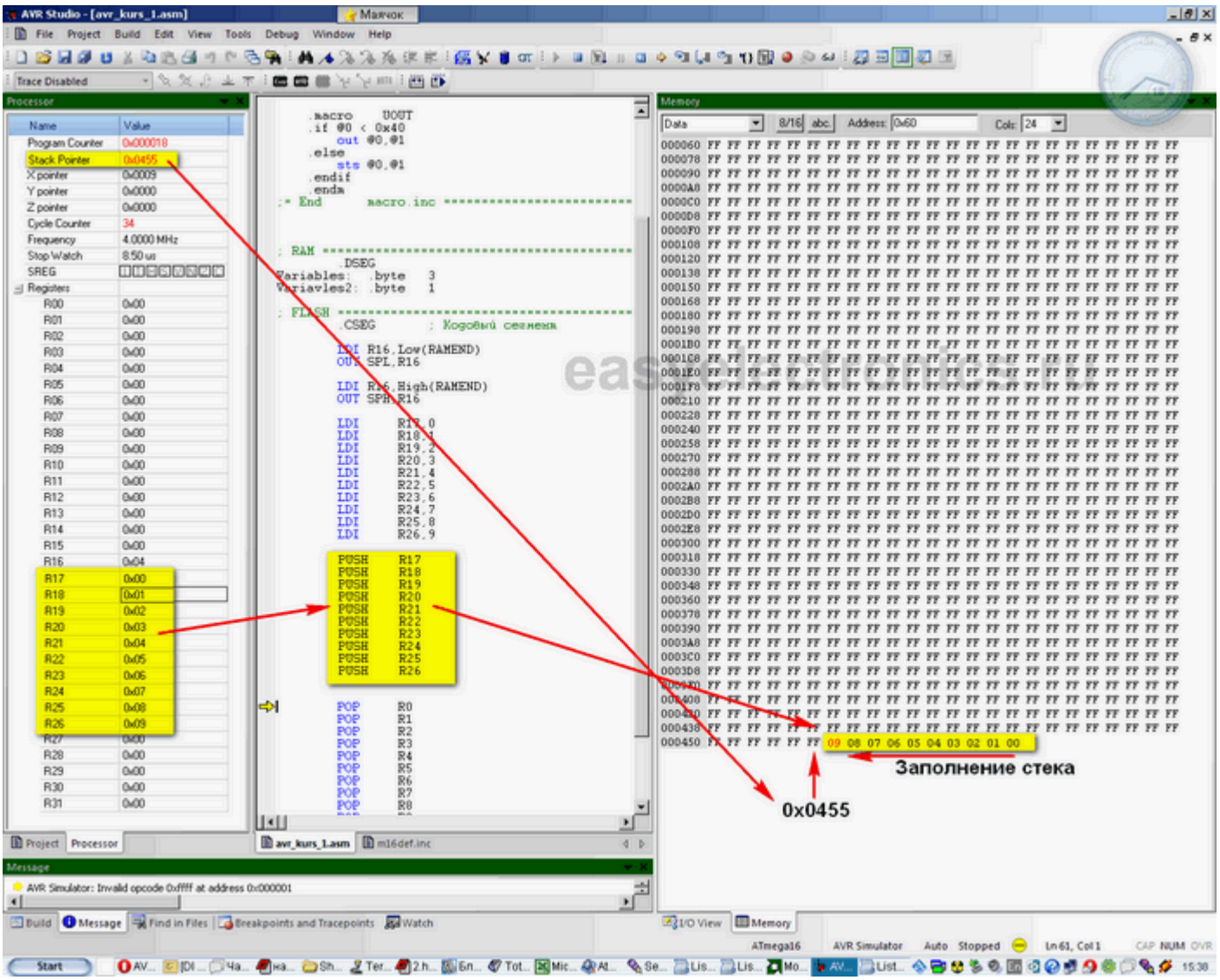
А теперь запускай студию в пошаговое выполнение и следи за тем как будет меняться SP. Stack Pointer можно поглядеть в студии там же, где и Program Counter.

Вначале мы инициализируем стек и загрузим регистры данными. В результате получится следующая картина:



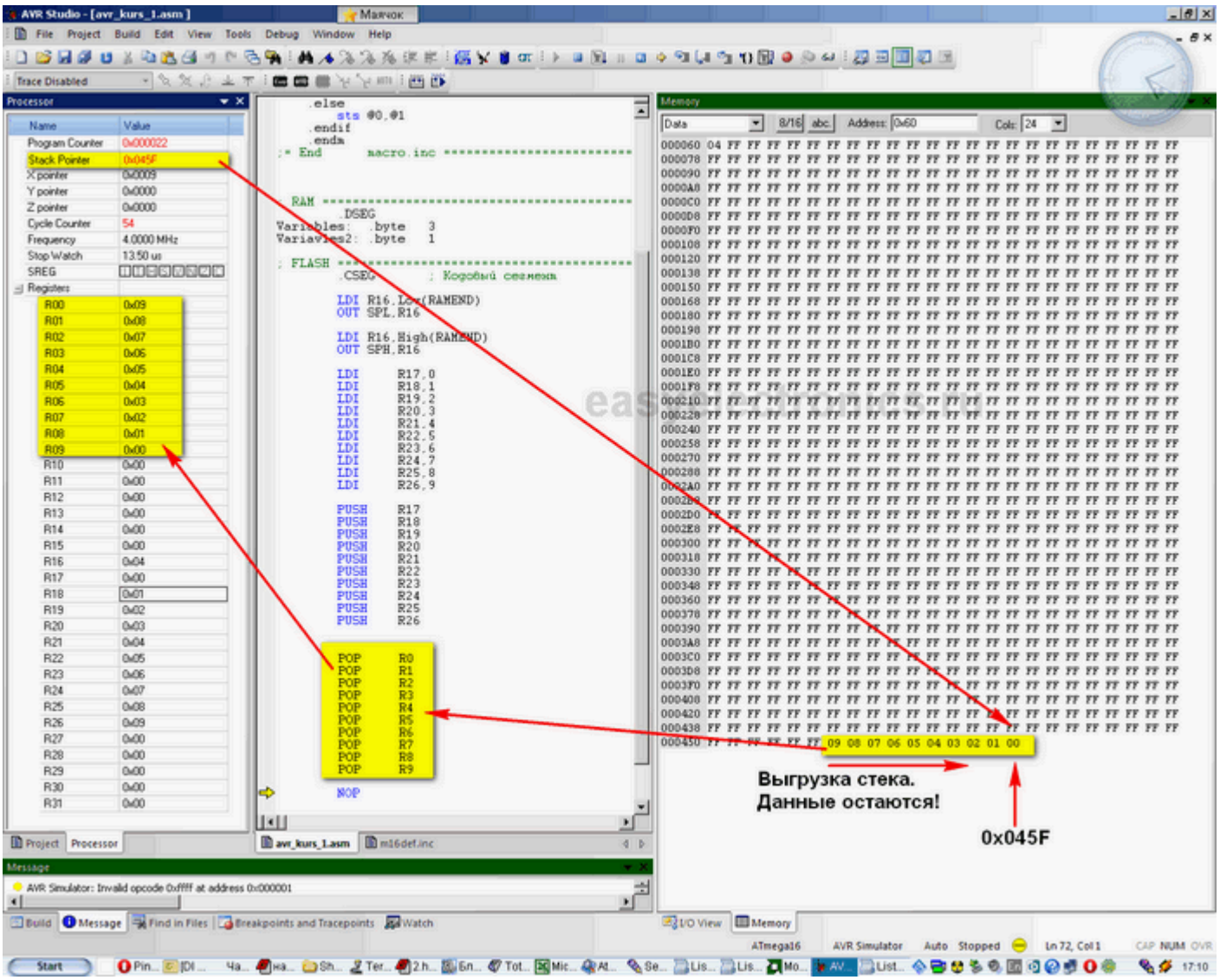
[увеличить](#) [2]

Затем начнем по одному пихать данные в стек. При этом будет видно, как данные заполняют память начиная от конца, к началу. А SP меняется в сторону уменьшения. Указывая на следующую ячейку.
После всех команд PUSH наши данные окажутся в памяти:



увеличить [3]

Дальше, командой POP, мы достаем данные из стека. Обрати внимание на то, что нам совершенно не важно откуда мы положили данные в стек и куда мы их будем сгружать. Главное порядок укладки! Ложили мы из старших регистров, а достанем в младшие. При этом указатель стека будет увеличиваться.



увеличить [4]

Да, еще немаловажный момент. Данные при этом никуда из памяти не деваются, так и остаются висеть в памяти. При следующем заполнении стека их просто перепишет и все.

Как пользоваться стеком?
Ну во первых, стек используют команды вызовов и возвратов (CALL, RCALL, ICALL, RET, RETI), а еще это удобное средство по быстрому свапить или сохранять байты.

Вот, например, надо тебе обменять содержимое двух регистров R17 и R16 местами. Как сделать это без использования третьего регистра? Самое простое — через стек. Положим из одного, достанем в другой.

```
1          PUSH    R16
2          PUSH    R17
3          POP     R16
4          POP     R17
```

Или сохранить какой нибудь значение регистра, пока его используем в другом месте.

Например, я уже говорил про ограничение младших РОН — они не дают записать в себя число напрямую. Только через регистры старшей группы. Но это же неудобно!

Проблема решается с помощью макроса. Я назвал его LDIL — LDI low

```
1         .MACRO LDIL
2         PUSH    R17     ; Сохраним значение одного из старших регистров в стек.
3         LDI     R17,@1  ; Загрузим в него наше непосредственное значение
4         MOV     @0,R17  ; перебросим значение в регистр младшей группы.
5         POP     R17     ; восстановим из стека значение старшего регистра.
6         .ENDM
```

Теперь можно легко применять нашу самодельную команду.

```
1         LDIL    R0,18
```

Со временем, файл с макросами обрастает такими самодельными командами и работать становится легко и приятно.

Стековые ошибки
Стек растет навстречу данным, а теперь представьте что у нас в памяти есть переменная State и расположена она по адресу, например, 0x0450. В опасной близости от вершины стека. В переменной хранится, например, состояние конечного автомата от которого зависит дальнейшая логика работы программы. Скажем если там 3, то мы идем делать одно, если 4 то другое, если 5 то еще что-то и так до 255 состояний. И по логике работы после 3 должна идти 4ре, но никак не 10

И вот было там 3. И тут, в один ужасный момент, условия так совпали, что стек разросся и его вершина дошла до этой переменной, вписав туда значение, скажем 20, а потом борзо свалила обратно. Оставив гадость — классический пример переполнения стека. И логика программы вся нахрен порушилась из-за этого.

Либо обратный пример — стек продавился до переменных, но в этот момент переменные обновились и перезаписали стековые данные. В результате, со стека снялось что-то не то (обычно кривые адреса возврата) и программе сорвало крышу. Вот такой вариант, кстати, куда более безобидный, т.к. в этом случае косяк видно сразу и он не всплывает ВНЕЗАПНО спустя черт знает сколько времени.

Причем эта ошибка может то возникать, то исчезать. В зависимости от того как работает программа и насколько глубоко она прогружает стек. Впрочем, такое западло чаще встречается когда пишешь на Си, где не видно насколько активно идет работа со стеком. На асме все гораздо прозрачней. И тут такое может возникнуть из-за откровенно кривого алгоритма.

На долю ассемблерщиков часто выпадают другие стековые ошибки. В первую очередь забычивость. Что то положил, а достать забыл. Если дело было в подпрограмме или в прерывании, то искажается адрес возврата (о нем чуть позже), стек срывает и прога мгновенно рушится. Либо невнимательность — сохранял данные в одном порядке, а достал в другом. Опа и содержимое регистров обменялось.

Чтобы избегать таких ошибок нужно, в первую очередь, следить за стеком, а во вторых грамотно планировать размещение переменных в памяти. Держа наиболее критичные участки и переменные (такие как состояния конечных автоматов или флаги логики программы) подальше от стековой вершины, поближе к началу памяти.

У некоторых возникнет мысль, что можно же взять и стек разместить не на самом конце ОЗУ, а где нибудь поближе, оставив за ним карман для критичных данных. На самом деле не слишком удачная мысль. Дело в том, что стек можно продавить как вниз, командой PUSH так и вверх — командами POP. Второе хоть и случается намного реже, т.к. это больше грех кривых рук, чем громоздкого алгоритма, но тоже бывает.
Но главное это то, что стек сам по себе сверхважная структура. На ней держится весь механизм подпрограмм и функций. Так что срыв стека это ЧП в любом случае.

Стековые извраты
Моя любимая тема. =)))) Несмотря на то, что стековый указатель сам вычисляется при командах PUSH и POP, никто не мешает нам выковырять его из SP, да использовать его значения для ручного вычисления адреса данных лежащих в стеке. Либо подправить стековые данные как нам угодно.
Зачем? Ну применений можно много найти, если напрячь мозг и начать думать нестандартно :))))

Плюс через стек, в классическом Си и Паскале на архитектуре x86 передаются параметры и работают локальные переменные. Т.е. перед вызовом функции вначале все переменные пихаются в стек, а потом, после вызова функции, в стек пихаются байты будущих локальных переменных.

После, используя SP как точку отсчета, мы можем обращаться с этими переменными как нам угодно. А при освобождении стека командой POP они аннигилируются, освобождая память.

В AVR все несколько не так (видимо связано с малым обьемом памяти, где в стек особо не насуешься, зато есть прорва РОН, но механизм этот тоже можно попробовать использовать.

Правда это уже напоминает нейрохирургию. Чуть ошибся и пациент труп.

Благодаря стеку и ОЗУ можно обходиться всего двумя-тремя регистрами, не особо испытывая напряг по поводу их нехватки.

Флеш память

Память EEPROM маленькая, всего считанные байты, а иногда нужно сохранить кучу данных, например, послание инопланетянам или таблицу синусов, чтобы не тратить время на ее расчет. Да мало ли что нужно заранее заныкать в памяти. Поэтому данные можно забивать в память программ, в те самые килобайты флеша, что имеет контроллер на борту.

Записать то мы запишем, а как достать? Для этого сначала надо туда что-либо положить.
Поэтому добавляй в конце программы, в пределах сегмента .CSEG метку, например, data и после нее, используя оператор .db, вписывай свои данные.

Оператор DB означает что мы на каждую константу используем по байту. Есть еще операторы задающий двубайтные константы DW (а также DD и DQ).

```
1 data:   .db     12,34,45,23
```

Теперь, метка data указывает на адрес первого байта массива, остальные байты находятся смещением, просто добавляя к адресу единичку.

Одна тонкость — дело в том, что адрес метки подставляет компилятор, а он считает его адресом перехода для программного счетчика. А он, если ты помнишь, адресует двубайтные слова — ведь длина команды у нас может быть либо 2 либо 4ре байта.

А данные у нас лежат побайтово и контроллер при обращении к ним адресует их тоже побайтово. Адрес в словах меньше в два раза чем адрес в байтах и это надо учитывать, умножая адрес на два.

Для загрузки данных из памяти программ используется команда из группы Load Program Memory

Например, LPM Rn,Z

Она заносит в регистр Rn число из ячейки на которую указывает регистровая пара Z. Напомню, что Z это два регистра, R30 (ZL) и R31 (ZH). В R30 заносится младший байт адреса, а в R31 старший.

В коде выглядит это так:

```
1          LDI     ZL,low(data*2) ; заносим младший байт адреса, в регистровую пару Z
2          LDI     ZH,high(data*2); заносим старший байт адреса, в регистровую пару Z
3                                         ; умножение на два тут из-за того, что адрес
4  указан в
5                                         ; в двубайтных словах, а нам надо в байтах.
6                                         ; Поэтому и умножаем на два
7                                         ; После загрузки адреса можно загружать число из
8  памяти
9
10         LPM     R16, Z                 ; в регистре R16 после этой команды будет число
11 12,
12                                        ; взятое из памяти программ.
13
```

```
; где то в конце программы, но в сегменте .CSEG
data:   .db     12,34,45,23
```

AVR. Учебный курс. Подпрограммы и прерывания

Подпрограммы
Когда один и тот же участок кода часто повторяется, то разумно как то его вынести и использовать многократно. Это дает просто колоссальный выйгрыш по обьему кода и удобству программирования.

Вот, например, кусок кода, передающий в регистр UDR байты с некоторой выдержкой, выдержка делается за счет вращения бесконечного цикла:

```
1          .CSEG
2          LDI R16,Low(RAMEND)     ; Инициализация стека
3          OUT SPL,R16             ; Обязательно!!!
4
5          LDI R16,High(RAMEND)
6          OUT SPH,R16
7
8          .equ    Byte    = 50
9          .equ    Delay   = 20
10
11         LDI     R16,Byte        ; Загрузили значение
12 Start:  OUT     UDR,R16         ; Выдали его в порт
13
14         LDI     R17,Delay       ; Загрузили длительность задержки
15 M1:     DEC     R17             ; Уменьшили на 1
16         NOP                     ; Пустая операция
17         BRNE    M1              ; Длительность не равна 0? Переход если не 0
18
19         OUT     UDR,R16         ; Выдали значение в порт
20
21         LDI     R17,Delay       ; Аналогично
22 M2:     DEC     R17
23         NOP
24         BRNE    M2
25
26         OUT     UDR,R16
27
28         LDI     R17,Delay
29 M3:     DEC     R17
30         NOP
31         BRNE    M3
32
33         RJMP    Start           ; Зациклим программу
```

Сразу напрашивается повторяющийся участок кода вынести за скобки.

```
1         LDI     R17,Delay
2 M2:     DEC     R17
3         NOP
4         BRNE    M2
```

Для этих целей есть группа команд перехода к подпрограмме CALL (ICALL, RCALL, CALL)
И команда возврата из подпрограммы RET

В результате получается такой код:

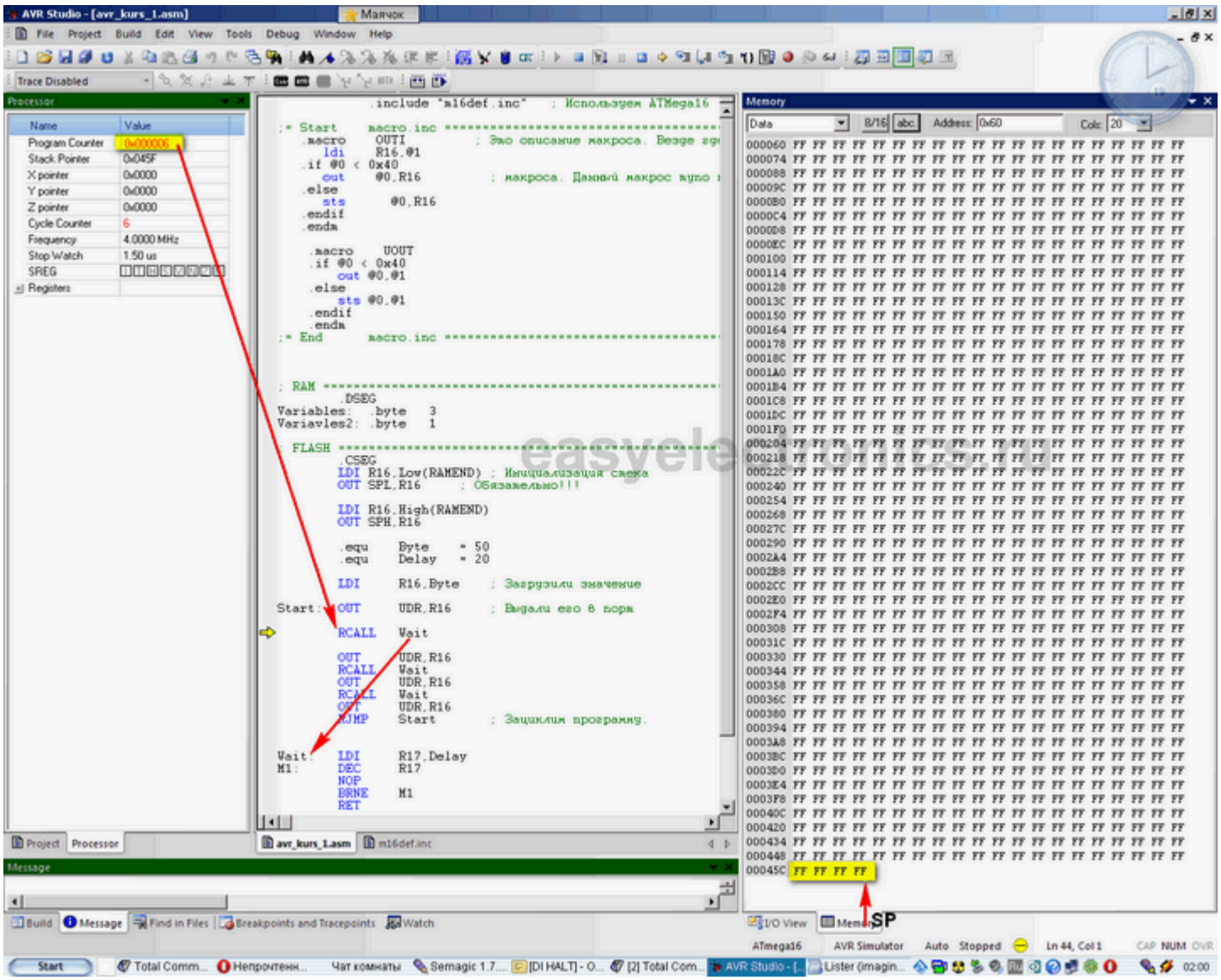
```
1          .CSEG
```

```
2          LDI R16,Low(RAMEND)    ; Инициализация стека
3          OUT SPL,R16            ; Обязательно!!!
4
5          LDI R16,High(RAMEND)
6          OUT SPH,R16
7
8          .equ    Byte    = 50
9          .equ    Delay   = 20
10
11         LDI     R16,Byte       ; Загрузили значение
12 Start:  OUT     UDR,R16        ; Выдали его в порт
13
14         RCALL   Wait
15
16         OUT     UDR,R16
17         RCALL   Wait
18         OUT     UDR,R16
19         RCALL   Wait
20         OUT     UDR,R16
21         RCALL   Wait
22         RJMP    Start          ; Зациклим программу.
23
24
25 Wait:   LDI     R17,Delay
26 M1:     DEC     R17
27         NOP
28         BRNE    M1
29         RET
```

Как видишь, программа резко сократилась в размерах. Теперь скопируй это в студию, скомпилируй и запусти на трассировку. Я хочу показать как работает команда RCALL и RET и при чем тут стек.

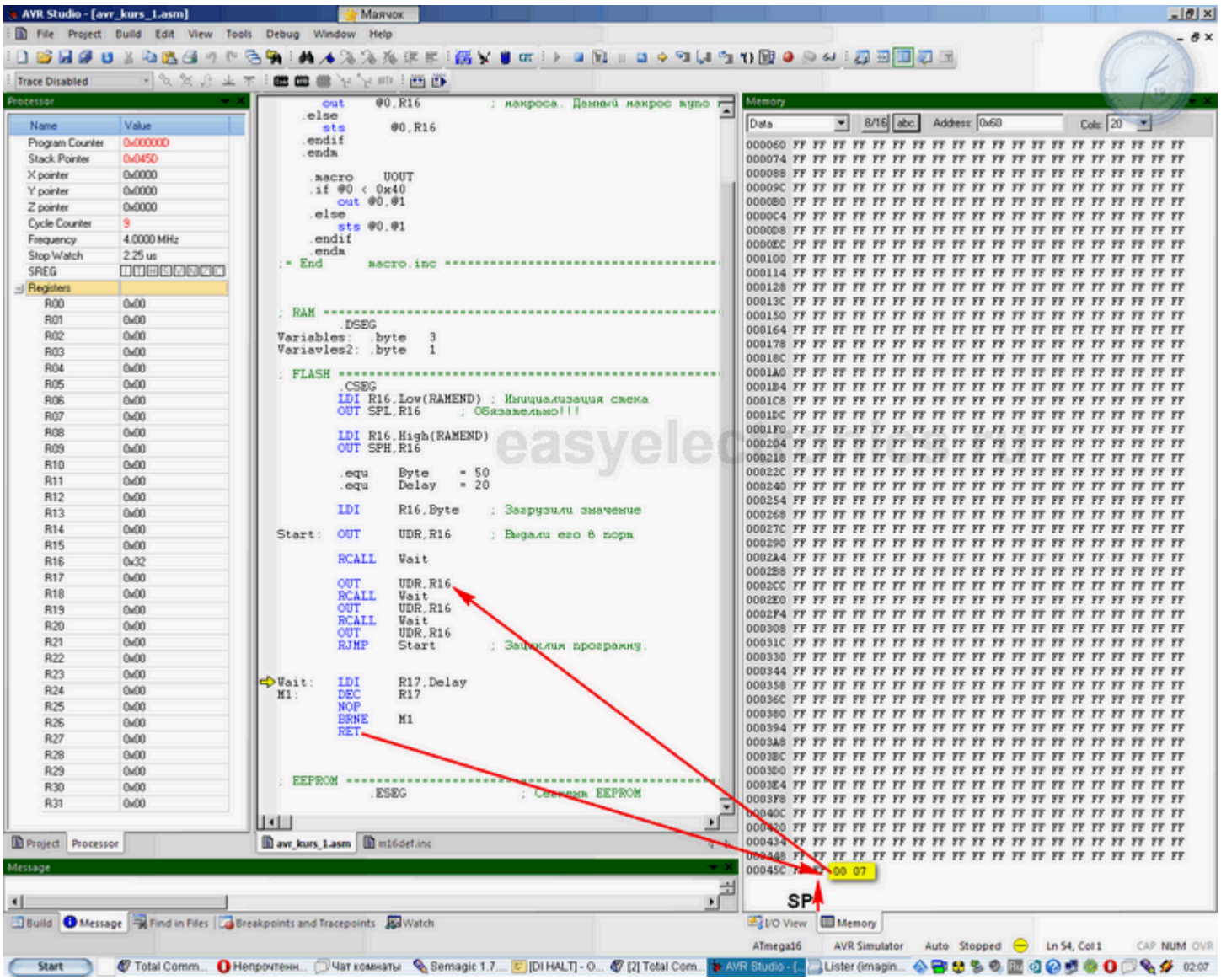
Вначале программа, как обычно, инициализирует стек. Потом загружает наши данные в регистры R16 и выдает первый байт в UDR… А потом по команде RCALL перейдет по адресу который мы присвоили нашей процедуре, поставив метку Wait в ее начале. Это понятно и логично, гораздо интересней то, что произойдет в этот момент со стеком.

До выполнения RCALL



Увеличить [1]
Адрес команды RCALL в памяти, по данным PC = 0×000006, адрес следующей команды (OUT UDR,R16), очевидно, будет 0×000007. Указатель стека SP = 0x045F — конец памяти, где ему и положено быть в этот момент.

После RCALL



Увеличить [2]
Смотри, в стек пихнулось число 0×000007, указатель сместился на два байта и стал 0x045D, а контроллер сделал прыжок на адрес Wait.

Наша процедура спокойно выполняется, как ей и положено, а по команде RET процессор достанет из стека наш заныченный адрес 0×000007 и прыгнет сразу же на команду OUT UDR,R16

Таким образом, где бы мы не вызвали нашу процедуру Wait — мы всегда вернемся к тому же месту откуда вызвали, точнее на шаг вперед. Так как при переходах в стеке сохраняется адрес возврата. А если испортить стек? Взять и засунуть туда еще что нибудь? Подправь процедуру Wait и добавь туда немного бреда, например, такого

```
1 Wait:   LDI     R17,Delay
2 M1:     DEC     R17
3         NOP
4         BRNE    M1
5
6         PUSH    R17             ; Ой, я не специально!
7
8         RET
```

Перекомпиль и посмотри что будет =) Заметь, компилятор тебе даже слова не скажет. Мол все путем, дерзай :)

До команды PUSH R17 в стеке будет адрес возврата 00 07, так как в регистре R17 ,в данный момент, ноль, и этот ноль попадет в стек, то там будет уже 00 00 07.

А потом идет команда RET…Она глупая, ей все равно! RET тупо возьмет два первых верхних байта из стека и запихает их в Programm Counter.

И куда мы перейдем? Правильно — по адресу 00 00, в самое начало проги, а не туда откуда мы ушли по RCALL. А будь в R17 не 00, а что нибудь другое и попади это что-то в стек, то мы бы перешли вообще черт знает куда с непредсказуемыми последствиями. Это и называется срыв стека.

Но это не значит, что в подпрограммах нельзя пользоваться стеком в своих грязных целях. Можно!!! Но делать это надо с умом. Класть туда данные и доставать их перед выходом. Следуя железному правилу «Сколько положил в стек — столько и достань!», чтобы на выходе из процедуры для команды RET лежал адрес возврата, а не черти что.

Мозговзрывной кодинг
Да, а еще тут возможны стековые извраты. Кто сказал, что мы должны вернуться именно туда откуда были вызываны? =))) А если условия изменились и по итогам вычислений в процедуре нам ВНЕЗАПНО туда стало не надо? Никто не запрещает тебе нужным образом подправить данные в стеке, а потом сделать RET и процессор, как миленький, забросит тебя туда куда надо. Легко!

Более того, я когда учился в универе и сдавал лабы по ассемблеру, то лихо взрывал мозги нашему преподу такими конструкциями (там, правда, был 8080, но разница не велика, привожу пример для AVR):

```
1         LDI     R17,low(M1)
2         PUSH    R17
3         LDI     R17,High(M1)
4         PUSH    R17
5
6 ; потом дофига дофига другого кода... для отвлечения
7 ; внимания, а затем, в нужном месте, ВНЕЗАПНО
8
9         RET
```

И происходил переход на метку M1, своего рода извратский аналог RJMP M1. А точнее IJMP, только вместо Z пары мы используем данные адреса загруженные в стек из любого другого регистра, иногда пригождается. Но без особой нужды таким извратом заниматься не рекомендую — запутывает программу будь здоров.

Но побалуйся обязательно, чтобы во всей красе прочувствовать стековые переходы.

Отлаженные и выверенные подпрограммы кода можно запихать в отдельный модуль и таскать их из проекта в проект, не изобретая каждый раз по велосипеду.

Иногда подпрограммы ошибочно называют функциями. Отличие подпрограммы от функции в том, что функция всегда имеет какое то значение на входе и выдает ответ на выходе, как в математике. Ассемблерная подпрограмма же не имеет таких механизмов и их приходится изобретать самому. Например, передавая в РОН или в ячейках ОЗУ.

Подпрограммы vs Макросы
Но не стоит маникально все повторяющиеся участки заворачивать в подпрограммы. Дело в том, что переход и возврат добавляют две команды, а еще у нас идет прогрузка стека на 2 байта. Что тоже не есть гуд. И если заменяется три-четыре команды, то овчинка с CALL-RET не стоит выделки и лучше запихать все в макрос.

Прерывания

Это аппаратные события. Ведь у микроконтроллера кроме ядра есть еще дофига периферии. И она работает параллельно с контроллером. Пока контроллер занимается вычислением или гоняет байтики по памяти — АЦП может яростно оцифровывать входное напряжение, USART меланхолично передавать или принимайть байтик, а EEPROMка неспеша записывать в свои тормозные ячейки очередной байт.

А когда периферийное устройство завершает свою работу оно поднимает флаг готовности. Мол, чувак, у меня все пучком, забирай результат. Процессор может проверить этот флаг в общем цикле и как то его обработать.

Но некоторые события в принципе не могут ждать, например USART — вовремя не обработаешь входящий байт, считай провафлил передачу, т.к. передающий девайс пошлет второй, ему плевать успел ты его обработать или нет. Для таких срочных дел есть прерывания.

У AVR этих прерываний с полтора десятка наберется, на каждое переферийное устройство по прерыванию, а на некотрые и не по одному. Например, у USART их целых три — Байт пришел, Байт ушел, Передача завершена.

Как это работает

Когда случается прерывание, то процессор тут же завершает текущую команду, пихает следующий адрес в стек (точно также как и при CALL) и переходит… А куда, собственно, он переходит?

А переходит он на фиксированный вектор прерывания. За каждым аппаратным прерыванием закреплен свой именной адрес. Все вместе они образуют таблицу векторов прерывания. Расположена она в самом начале памяти программ. Для Атмега16, используемой в Pinboard [3] таблица прерываний выглядит так:

```
1  RESET           0x0000  ; Reset Vector
2  INT0addr        0x0002  ; External Interrupt Request 0
3  INT1addr        0x0004  ; External Interrupt Request 1
4  OC2addr         0x0006  ; Timer/Counter2 Compare Match
5  OVF2addr        0x0008  ; Timer/Counter2 Overflow
6  ICP1addr        0x000a  ; Timer/Counter1 Capture Event
7  OC1Aaddr        0x000c  ; Timer/Counter1 Compare Match A
8  OC1Baddr        0x000e  ; Timer/Counter1 Compare Match B
9  OVF1addr        0x0010  ; Timer/Counter1 Overflow
10 OVF0addr        0x0012  ; Timer/Counter0 Overflow
11 SPIaddr         0x0014  ; Serial Transfer Complete
12 URXCaddr        0x0016  ; USART, Rx Complete
13 UDREaddr        0x0018  ; USART Data Register Empty
14 UTXCaddr        0x001a  ; USART, Tx Complete
15 ADCCaddr        0x001c  ; ADC Conversion Complete
16 ERDYaddr        0x001e  ; EEPROM Ready
17 ACIaddr         0x0020  ; Analog Comparator
18 TWIaddr         0x0022  ; 2-wire Serial Interface
19 INT2addr        0x0024  ; External Interrupt Request 2
20 OC0addr         0x0026  ; Timer/Counter0 Compare Match
21 SPMRaddr        0x0028  ; Store Program Memory Ready
```

Как видишь, это первые адреса флеша. На каждый вектор отводится по два байта в которые мы можем записать любую команду. Но что попало туда обычно не вписывают, а ставят сразу JMP на какую нибудь метку, где уже можно спокойно сделать все что душе угодно, не стесняясь размеров кода.

Запишем эту бодягу в цивильной форме, через ORG и добавим немного кода.

```
1           .CSEG
2           .ORG $000         ; (RESET)
3           RJMP   Reset
4           .ORG $002
5           RETI              ; (INT0) External Interrupt Request 0
6           .ORG $004
7           RETI              ; (INT1) External Interrupt Request 1
8           .ORG $006
9           RETI               ; (TIMER2 COMP) Timer/Counter2 Compare Match
10          .ORG $008
11          RETI              ; (TIMER2 OVF) Timer/Counter2 Overflow
12          .ORG $00A
13          RETI               ; (TIMER1 CAPT) Timer/Counter1 Capture Event
14          .ORG $00C
```

```
15          RETI             ; (TIMER1 COMPA) Timer/Counter1 Compare Match A
16          .ORG $00E
17          RETI             ; (TIMER1 COMPB) Timer/Counter1 Compare Match B
18          .ORG $010
19          RETI             ; (TIMER1 OVF) Timer/Counter1 Overflow
20          .ORG $012
21          RETI             ; (TIMER0 OVF) Timer/Counter0 Overflow
22          .ORG $014
23          RETI             ; (SPI,STC) Serial Transfer Complete
24          .ORG $016
25          RJMP   RX_OK     ; (USART,RXC) USART, Rx Complete
26          .ORG $018
27          RETI             ; (USART,UDRE) USART Data Register Empty
28          .ORG $01A
29          RETI             ; (USART,TXC) USART, Tx Complete
30          .ORG $01C
31          RETI              ; (ADC) ADC Conversion Complete
32          .ORG $01E
33          RETI             ; (EE_RDY) EEPROM Ready
34          .ORG $020
35          RETI             ; (ANA_COMP) Analog Comparator
36          .ORG $022
37          RETI             ; (TWI) 2-wire Serial Interface
38          .ORG $024
39          RETI             ; (INT2) External Interrupt Request 2
40          .ORG $026
41          RETI             ; (TIMER0 COMP) Timer/Counter0 Compare Match
42          .ORG $028
43          RETI             ; (SPM_RDY) Store Program Memory Ready
44
45          .ORG   INT_VECTORS_SIZE      ; Конец таблицы прерываний
46
47 ;---------------------------------------------------------------------
48 ; Это обработчик прерывания. Тут, на просторе, можно наворотить сколько
49 ; угодно кода.
50 RX_OK:   IN     R16,UDR        ; Тут мы делаем что то нужное и полезное
51
52          RETI                  ; Прерывание завершается командой RETI
53 ;---------------------------------------------------------------------
54
55
56 Reset:  LDI R16,Low(RAMEND)    ; Инициализация стека
57          OUT SPL,R16           ; Обязательно!!!
58
59          LDI R16,High(RAMEND)
60          OUT SPH,R16
61
62          SEI                   ; Разрешаем прерывания глобально
63          LDI   R17,(1<<RXCIE)  ; Разрешаем прерывания по приему байта
64          OUT    UCSRB,R17
65
66 M1:      NOP
67          NOP
68          NOP
69          NOP
70          RJMP M1
```

Теперь разберем эту портянку. Контроллер стартует с адреса 0000, это точка входа. Там мы сразу же делаем бросок на метку RESET. Если это не сделать, то контроллер пойдет выполнять команды из таблицы векторов, а они не для того там посажены. Да и не далеко он ускачет — без инициализации стека и наличия адреса возврата в нем первый же RETI вызовет коллапс. Ведь RETI работает почти также как и RET.

Поэтому сразу уносим оттуда ноги на RESET. Где первым делом инициализируем стек. А потом, командой SEI, разрешаем прерывания. И установкой бита в регистре периферии UCSRB включаем прерывание по приему байта.

Дальше зацикливаемся и ждем когда в приемный буффер USART извне свалится байт. Запускай это дело в эмуляцию и начинай трассировать по одной команде. Сначала, как я и говорил, проц прыгнет на метку Reset, потом выставит нужные значения и наглухо зациклится на

```
1 M1:       NOP
2           NOP
3           NOP
4           NOP
5           RJMP M1
```

До прихода байта. Но как же нам осуществить этот приход байта если весь наш эксперимент не более чем симуляция виртуального процессора в отладчике? А очень просто!

Вручную вписать этот байт в регистр и вручную же протыкать флаг, словно байт пришел. За прием байта отвечает флаг RXC в регистре периферии UCSRA, раздел USART. Найди там бит RXC и тыкни его, чтобы закрасился. Все, прерывание вроде как наступило.

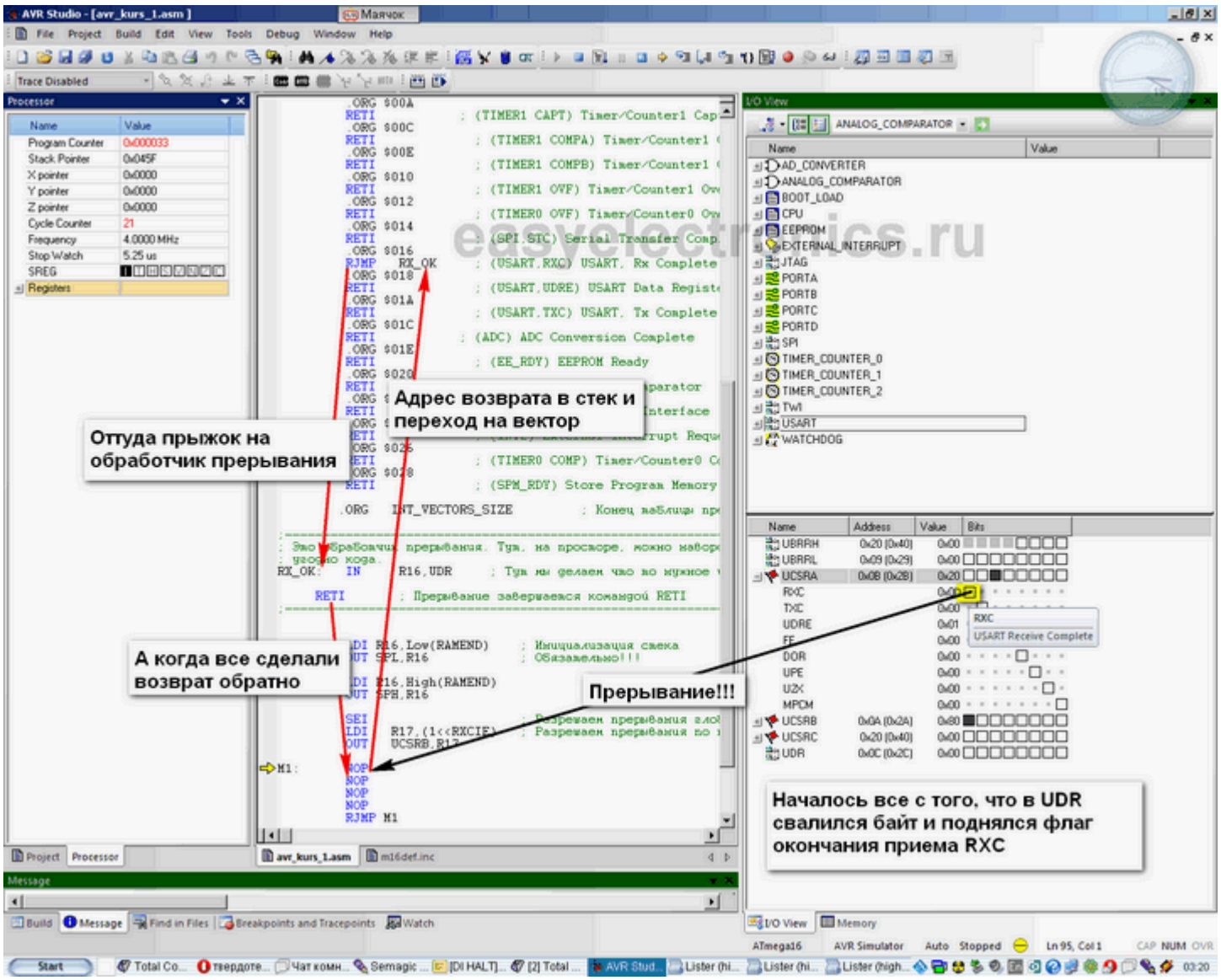
Нажми F11, чтобы сделать еще один шаг по программе… Опа, стрелочка улетела в таблицу векторов, как раз на вектор

```
1         .ORG $016
2         RJMP   RX_OK     ; (USART,RXC) USART, Rx Complete
```

А оттуда уже прыжок сразу же на метку RX_OK, где мы забираем данные из регистра UDR в R17 и выходим из прерывания, по команде RETI.

При этом процессор вернется в наш бесконечный цикл, в то место откуда прерывался.

Вот, как это было, если по коду:



Увеличить [4]

Вот, вроде теперь вопросов по выполнению быть не должно.

Разрешение и запрещение прерываний
Прерываний много, но по умолчанию они все запрещены. Во-первых, глобально — есть в процессоре в регистре SREG (о нем чуть позже) флаг I (interrupt) когда он равен 0, то все прерывания запрещены вообще, глобально, все без исключения.

Когда он равен 1, то прерывания глобально разрешены, но могут быть запрещены локально.

Устанавливается и сбрасывается этот флаг командами

- SEI — разрешить прерывания
- CLI — запретить прерывания (Да, по поводу моего ника, DI это, кстати, то же самое что и CLI, но для процессора Z80 ;))

Кроме того, у каждого прерывания есть еще свой собственный бит локального разрешения. В примере с UDR это бит RXCIE (Receive Complete Interrupt Enable) и расположены они в портах ввода вывода

По дефолту, после старта, все прерывания запрещены локально и глобально. И их надо разрешать вручную, не забыв прописать обработчик прерывания.

Если не прописать обработчик и по ошибке разрешить прерывания, то когда это ошибочное прерывание вдруг сработает, то процессор ускачет по вектору, а вот там все зависит от того, что ты туда прописал.

Если все неиспользуемые прерывания заглушены командой RETI то ничего не пройзойдет — как пришел так и вернется обратно. Если же там ничего нет, то процессор будет выполнять эту пустоту пока не доберется до живого кода. Это может быть как переход на обработчик другого прерывания (ниже по таблице векторов) так и начало основного кода, тело обработчика прерывания, какая либо процедура, записанная до метки start. Да что угодно, что первой попадется.

Сам понимаешь, что в таком случае ни о какой корректной работе говорить не придется. Кстати, почти все сишные компиляторы, неиспользуемые обработчики глушат на RESET т.е. случайный вызов несуществующего прерывания приводит к сбросу.

С одной стороны это бага, с другой фича — так называемая ситуация максимизации ошибки. Т.е. большой фатальный глюк, обрушивающий всю программу нахрен, куда безопасней мелкой ошибки потому, что вылазит сразу же. А ошибка может поджидать годами и спокойно пересидеть период отладки, а вылезти уже в готовом устройстве. И непременно перед тем как заказчик выпишет тебе бабло за разработку :)

Итак, повторюсь для ясности. Прерывания по дефолту запрещены локально и глобально. Разрешают их по мере надобности. Причем делают это дважды — на локальном уровне и на глобальном.

О том, что прерывание произошло извещает флаг этого прерывания в регистре ввода вывода. Пока флаг поднят считается что прерывание не обработано и процессор будет пытаться по этому прерыванию перейти (если конечно прерывания разрешены и локально, и глобально).

Флаг этого события сбрасывается либо сам, при переходе к обработчику прерывания, либо при совершении какого-либо действия. Например, чтобы сбросить флаг события прерывания RxC надо считать байт из UDR. Протрассируй программу и сам увидишь, что сброс флага RxC происходит после выполнения команды

```
1 IN R16,UDR
```

Либо флаг скидывают вручную — **записью в этот флаг единицы. Не нуля! Единицы!** Подробней в даташите на конкретную периферию.

Очередность прерываний
Но что будет если произошло одно прерывание и процессор ушел на обработчик, а в этот момент произошло другое прерывание?

А ничего не будет, при **уходе на прерывание происходит аппаратный глобальный запрет всех других прерываний**— просто сбрасывается флаг I, а по команде RETI, при возврате, этот флаг устанавливается в 1. В этом, кстати, отличие RET от RETI

Но! Никто не запрещает нам внутри обработчика поставить SEI и разрешить прерывания. При этом мы получим вложенные прерывания. Можно, но это опасно. Черевато переполнением стека и прочими гадостями. Так что это надо делать с твердым осознанием последствий.

Что тогда? Прерывание которое пришло во время обработки первого потеряется?

Нет, не потеряется. Флаг то его события никто сам не снимет, так что только процессор выйдет из первого обработчика (разреша при этом глобальные прерывания), как это не снятый флаг сгенерирует новое прерывание и процессор его обработает.

А теперь такая ситуация — прерывания запрещены, неважно по какой причине, и тут приходит два прерывания. Поднимает каждая свой флажок и оба ждут глобального разрешения. SEI!!! Кто пойдет первым? А первым пойдет то прерывание, чей вектор меньше по адресу, ближе к началу памяти. За ним второй.

В случае когда пришло несколько прерываний одного типа. Скажем, пока мы там ковырялись в обработчике, нам сбоку тут еще таймер три раза тикнул своим флагом, то обработается только одно событие, остальные могут потеряться.

Бег по граблям
Прерывания штука мощная, но опасная. С их помощью плодятся такие глюки, по сравнению с которыми стековые срывы так — семечки.

Вся засада багов из-за кривых прерываниях в том, что их практически невозможно отследить в отладчике.

Возникает плавающий глюк, появление которого зависит от того в каком именно месте кода произойдет вызов прерывания. Что поймать, сам понимаешь, почти невозможно.

Так что если у МК то понос, то золотуха, то программа петухом поет, а то молчит как рыба — знай, в 95% копать собаку надо в районе прерываний и их обработчиков.

Но если правильно написать прерывание, то багов оно не даст. Главное понимать в чем его опасность.

Грабли первые — спасай регистры!!!
Прерывание, когда оно разрешено, вызывается ВНЕЗАПНО, между двумя произвольными инструкциями кода. Поэтому очень важно, чтобы к моменту когда мы из прерывания вернемся все осталось как было.

Все регистры, используемые в обработчике прерываний, должны быть предварительно сохранены. Также должен быть сохранен регистр флагов SREG, в котором хранится результат логических операций. Результаты проще всего сохранять в стеке.

Приведу пример: Вот есть у нас обработчик прерывания который сравнивает байт на входе в USART и если он равен 10, выдает обратно отклик ‘t’ (ten в смысле).

```
1  RX_OK:
2
3          IN      R16,UDR
4          CPI     R16,10
5          BREQ    Ten
6          RJMP    Exit
7
8  Ten:    LDI     R17,'t'
9          OUT     UDR,R17
10
11 Exit:
```

И у нас есть код. Код скопипащен от балды из какого то проекта, даже пояснять не буду, не в нем суть.

```
1           . . .
2           LPM     R18,Z
3           CPI     R18,0
4
5           BREQ    ExitStr
6
7           SUBI    R16,65
8           LSL     R16
9
10          LDI     ZL,Low(Ltrs*2)
11          LDI     ZH,High(Ltrs*2)
12
13          ADD     ZL,R16
14          ADC     ZH,R1
15
16          . . .
```

Согласно идеи прерывания, наш обработчик может воткнуться между двумя любыми инструкциями.
Например, так:

```
1           . . .
```

```
2           LPM    R18,Z
3           CPI    R18,0
4
5           BREQ   ExitStr
6
7           SUBI   R16,65
8  >>>>>>>>>>>Прерывание >>>>>>>>>>>
9  RX_OK:
10
11         IN     R16,UDR
12         CPI    R16,10
13         BREQ   Ten
14         RJMP   Exit
15
16 Ten:    LDI    R17,'t'
17         OUT    UDR,R17
18
19 Exit:   RETI
20 <<<<<<<<<<< Возврат <<<<<<<<<<<<
21          LSL    R16
22
23          LDI    ZL,Low(Ltrs*2)
24          LDI    ZH,High(Ltrs*2)
25
26          ADD    ZL,R16
27          ADC    ZH,R1
28          . . .
```

До входа в прерывание, после команды SUBI R16,65 в R16 было какое то число из которого вычли 65.
И дальше с ним должны были провернуть операцию логического сдвига LSL R16, но тут вклинился обработчик, где в R16 записалось значение из UDR.

И мы выпали из обработчика перед командой LSL R16, но в R16 уже мусор какой то. Совершенно не те данные, что мы планировали.

Естественно вся логика работы от такого издевательства порушилась и возник глюк. Но стоит прерыванию прийти чуть раньше, на одну микросекунду — как глюк исчезнет, т.к. SUBI R16,65 будет уже после прерывания и логика не будет порушена, но может возникнуть другой глюк.

Полная лотерея, повезет не повезет. Может вылезти сразу, а может и через год идеальной работы, а потом также сгинет и сиди чеши репу что же это было — сбой по питанию, бага, или таракан по плате пробежал неудачно.

Чтобы такого не было в обязательно порядке надо сохранять регистры и SREG на входе в прерывание и доставать на выходе.

У нас тут, в обработчике, используется R17 и R16 и SREG (в него помещается результат работы команды CPI). Вот их и сохраним.

Выглдеть это будет так:

```
1  RX_OK:  PUSH    R16             ; Сохранили R16
2          IN      R16,SREG        ; Достали SREG в R16
3          PUSH    R16             ; Утопили его в стеке
4          PUSH    R17             ; Туда же утопили R17
5
6  ; Теперь можно со спокойной совестью работу работать.
7
8          IN      R16,UDR
9          CPI     R16,10
10         BREQ    Ten
11         RJMP    Exit
12
```

```
13 Ten:    LDI     R17,'t'
14         OUT     UDR,R17
15
16 ; А на выходе вернем все как было.
17 ; Достаем в обратном порядке
18
19 Exit:   POP     R17
20         POP     R16
21         OUT     SREG    R16
22         POP     R16
23         RETI                    ; Спокойно выходим. Регистры вернул как было.
```

Как же выносить данные из прерываний? Да по разному, можно в память сохранять, можно для этого спец регистр заиметь и знать, что он может в любой момент измениться в прерывании.

Грабли вторые — не тормози!!!
Прерывания отвлекают процессор от основных дел, более того, они блокируют другие прерывания. Поэтому в прерывании главное все сделать максимально быстро и свалить. Никаких циклов задержки, никаких долгоиграющих процедур. Никаких ожиданий аппаратного события. СКОРОСТЬ! СКОРОСТЬ! СКОРОСТЬ! Вот что должно тобой руководить при написании обработчика.

Заскочил — сделал — выскочил!

Но такая красивая схема возможна далеко не всегда. Иногда бывает надо по прерыванию делать и медленные вещи. Например, прием байтов из интерфейса и запись их в какую нибудь медленную память, вроде EEPROM. Как тогда быть?

А тут делают проще. Цель прерывания — во что бы то ни стало среагировать на событие именно в тот момент, когда оно пришло, чтобы не прозевать. А вот обрабатывать его прямо сейчас обычно и не обязательно. Заскочил в обработчик, схватил быстро тухнущие данные, перекинул их в буффер где им ничего не угрожает, поставил где-нибудь флажок отметку «мол там байт обработать надо» и вышел. Все! Остальное пусть сделает фоновая программа в порядке общей очереди.

Либо, если иначе никак нельзя, разрешай прерывания внутри обработчика, но смотри чтобы не возникли рекурсивные прерывания, когда один и тот же обработчик разрешает прерывания сам себе, образуя множественные вложенные вызовы с зарыванием в стек.

Грабли третьи — атомарный доступ
Есть ряд операций которые должны выполняться неразрывно. Например, чтение 16ти разрядных регистров таймера.

Ядро то у нас восьми-разрядное, поэтому 16ти разрядный регистр таймера считывается в два приема сначала младший байт, а потом старший. И вот в этом месте нельзя ни в коем случае допускать, чтобы между считыванием младшего и старшего было прерывание.

Т.к. после выхода из прерывания таймер уже может много чего натикать и информация уже будет не актуальная.

Поэтому перед чтением делаем CLI, а после SEI.

Также нельзя разрешать прерывания если одна и та же многоходовая операция делается и в прерывании и в главном цикле.

Например, прерывание хватает байты из АЦП и пишет их в буффер (ОЗУ), образуя связные цепочки данных. Главный же цикл периодически из этого буффера данные читает. Так вот, на момент чтения буффера из памяти, надо запрещать прерывания, чтобы никто не посмел в этот буффер что-нибудь записать.
Иначе мы можем получить невалидные данные — половина байтов из старого замера, половина из нового.

Причем, во многих случаях не обязательно глобально блокировать все прерывания вообще, достаточно заблокировать то локальное прерывание, которое может нам нагадить.

Грабли четвертые — не блокируй!!!
Третие грабли нужно внимательно обходить, но в паранойю впадать тоже не следует. Тупо запрещать прерывания везде, где мерещится бага, не стоит. Иначе можно прозевать события, а это плохо. Нет, обработчик то выполнится, но будет это уже не актуально — хороша ложка к обеду.

Обязательно погоняй в отладчике код (да хотя бы кучу NOP) с разными прерываниями. Чтобы понять и прочувствовать как ведут себя прерывания.

Старая версия статьи. Чисто поржать :) [5]

AVR. Учебный курс. Флаги и условные переходы

Есть в AVR (да и, пожалуй, во всех остальных процессорах) особый регистр **SREG**. О нем я несколько раз упоминал в прошлых статьях, но не вдавался в подробности. Чтож, пришло время рассказать, что же это же **SREG** такой и зачем он нужен.

SREG это регистр состояния ядра. Он так называется **S**tatus **Reg**ister. В этом регистре находится независимых битов — флажков. Которые могут быть либо 1 либо 0, в зависимости от выполненных в прошлом операций.

И вот по тому какие флаги стоят, можно понять что произошло с процессором и что нам дальше делать.

Например, если флаг **Z** (Zero) выставлен в 1, значит в ходе вычисления предыдущей математической операции в результате образовался ноль.

А если выставлен флаг **C** (Carry — заем, перенос), то мы из меньшего числа отняли большее, или же прибавили такое число, что результат стал больше 255.

А теперь подробней по каждому флагу.

- **I** — флаг разрешения прерываний. Когда установлен в 1 — прерывания разрешены.
- **T** — пользовательский флаг. Можно юзать по своему назначению.

  Кроме того, есть две команды которые позволяют в этот бит записать любой бит любого из 32 регистров общего назначения R0-R31 (далее буду их звать РОН). Это команды **BLD Rn,bit** и **BST Rn,bit**

- **H** — флаг полупереноса. Это если произошел заем бита из старшей половины байта в младшую. То есть когда из числа 0001 0111 пытаются вычести 0000 1000, то происходит заем бита из более старшего разряда, так как младшая тетрада уменьшаемого меньше чем младшей тетрады вычитаемого. Используется этот флаг в некоторых математических операциях.
- **S** — флаг знака. 1 — значит минус. При вычислении чисел со знаком он возникает если после арифметической операции возник отрицательный результат. Флаг S = V XOR N.
- **V** — Флаг переполнения дополнительного кода. Это если мы считаем число в дополнительном коде со знаком и оно вылезло за пределы регистра.

  Число в дополнительном коде с заемом — самая естественная форма представления числа во вселенной! Вот возьмем, например, число -61 как его получить? Ну не знаем мы про отрицательные числа! Просто! Вычтем его из нуля 00 — 61 = 39 Во как! Заняли из старшего разряда! Не похоже, да? Хорошо, проверим столбиком:
  61
  +
  39
  —
  00 А единичка не влезла в разрядность!

  ЧТД!!! (с) Лохов С.П. (Наш преподаватель по ассемблеру в универе)

  Вот двоичный доп код работает точно по такому же принципу. А в процессоре есть команды для перевода числа из в доп код за одну команду.

- **N** — флаг отрицательного значения. Если в результате арифметической операции 7 бит результата стал 1, то этот флаг тоже станет 1.
- **Z** — флаг нуля. Если в результате какой либо операции получился ноль, то вылазит этот флаг. Чертовски удобно для всех целей!
- **C** — флаг переноса. Если в результате операции произошел выход за границы байта, то выставляется этот флаг. Вообще самый юзаемый флаг после Z и I. Что только с ним не творят, как только не юзают.

Флаги, кроме автоматической установки, можно устанавливать и сбрасывать вручную. Для этого есть команды

SE* для установки и CL* для сброса. Вместо звездочки подставляется нужный флаг, например, CLI — запрет прерываний.

В даташите, в разделе Instruction Set Summary, написано какая команда что делает и на какие флаги влияет.

Пример:

```
1 INC     Rd      Increment                     Rd = Rd + 1            Z,N,V
2 DEC     Rd      Decrement                     Rd = Rd - 1            Z,N,V
3 TST     Rd      Test for Zero or Minus                Rd = Rd AND Rd          Z,N,V
4 CLR     Rd      Clear Register                Rd = Rd XOR Rd  Z,N,V
5 SER     Rd      Set Register                  Rd = $FF               None
```

Команда INC прибавляет к регистру 1, но несмотря на то, что она может добить регистр до переполнения, флаг переполнения C она не ставит. Такая особенность.

Зато она ставит флаги нуля, отрицательного значения и переполнения доп кода. Как инкремент может дать нуль? Элементарно, прибавив к -1 +1. Минус 1 в двоичной знаковой арифметике = FF. FF+1=1 00 ,но 1 не влезла в разрядность, поэтому 00 Логично же? :)

Или хотим мы, например, узнать в каком регистре число больше в R17 или R18

Нет ничего проще — к нашим услугам команда CP (нет, не детское порно, а Compare). Эта команда, у себя в уме, из R17 вычитает R18 при этом содержимое регистров не меняется, а меняются только флаги. И если, например, выскочил флаг C, значит при R17-R18 произошел заем, а значит R18 больше R17. Если флаг C не появился, то значит R17 больше чем R18. А коли у нас выскочил нуль, то значения равны. Есть еще команда CPI Rn,### работает также, но сравнивает регистр (только старшую группу) с произвольным числом. Используется чаще.

```
1         CP R17,R18
```

А потом смотрим флаги. И дальше в ход вступают команды условных переходов из группы BRANCH (ветвление). BR**

И вот в этом месте товарищей из ATMEL надо хватать за ноги и бить головой об стену. Потому что я не понимаю на кой хрен было так все запутывать, изобретая команды которых реально нет?

Суди сам. Флагов у нас 8, соответственно должно быть 16 возможных бранчей. 8 по условию — флаг есть, 8 по условию — флага нет. Бранчевых команд же у нас аж 20 штук.

```
1  BRBC #         переход если бит # в регистре SREG=0
2  BRBS #         переход если бит # в регистре SREG=1
3
4  BRCS           переход если C=1
5  BRCC           переход если C=0
6
7  BREQ           переход если Z=1
8  BRNE           переход если Z=0
9
10 BRSH           переход если C=0
11 BRLO           переход если C=1
12
```

```
13 BRMI            переход если N=1
14 BRPL            переход если N=0
15
16 BRGE            переход если S=0
17 BRLT            переход если S=1
18
19 BRHC            переход если H=0
20 BRHS            переход если H=1
21
22 BRTC            переход если T=0
23 BRTS            переход если T=1
24
25 BRVS            переход если V=1
26 BRVC            переход если V=0
27
28 BRID            переход если I=0
29 BRIE            переход если I=1
```

Однако, если глубоко копнуть, поглядеть на реальные коды команд, то окажется, что BRCS=BRLO и BRCC=BRSH — у них вообще одинаковый код.

А таких команд как BRBS # и BRBC # вообще не существует. Это всего лишь иносказательная запись всех остальных бранчей, в зависимости от номера бита #.

А таких команд синонимов там дофига :)
Так что гордое «131 Powerful Instructions – Most Single-clock Cycle Execution» на самом деле является не более чем гнилым маркетинговым высером. Нет там 131 команды! Есть 131 мнемоника, а это несколько разные вещи. Потому как ты помидор не обзови — картошкой он от этого не станет.

Правда, возможным оправданием такого поведения может служить то, что, дескать, так удобней — в зависимости от ситуации подставлять ту мнемонику которая написана более логично.

Может быть, но как по мне — только ситуацию запутывают. Из-за этого я после ассемблера C51 долго плевался на систему команд AVR, потом привык и ничо так, вкатило.

Итак, как работает любой из бранчей (да, тому кто придумал переименовать родимый J** в BR** я тоже ежедневно посылаю лучи поноса, надеюсь в сортире он себе уже кабинет обустроил).

Проверяется условие и если оно верное делается переход.

Например, на входе у нас значение.

- Если значение = 1, то делаем действие А
- Если значение = 2, то делаем действие Б
- А если значение меньше 13, то делаем действие ЦЭ
- Во всех остальных случаях не делаем ничего

Нет ничего проще, пусть значение приходит через регистр R16

```
1                CPI     R16,1          ; Сравниваем R16 с 1
2                BREQ    ActionA        ; Переход если равно (EQual, Z=1)
3                                       ; Если не равно, то идем дальше
4
5                CPI     R16,2          ; Сравниваем R16 с 2
6                BREQ    ActionB        ; Переход если равно
7                                       ; Если не равно, то идем дальше
8
9                CPI     R16,13         ; Сравниваем R16. т.е. R16-13
10               BRCS    ActionC        ; Если возник перенос, значит R16 меньше 13
11                                      ; Если не возник - идем дальше
12               RJMP    NoAction       ; Переход на выход
```

```
13
14 ActionA:        NOP
15                 NOP                    ; Делаем наш экшн
16                 NOP
17                 RJMP    NoAction       ; Выходим, чтобы не влезть в ActionB
18
19
20 ActionB:        NOP
21                 NOP                    ; Делаем наш экшн
22                 NOP
23                 RJMP    NoAction       ; Выходим, чтобы не влезть в ActionC
24
25
26 ActionC:        NOP
27                 NOP                    ; Делаем наш экшн
28                 NOP
29
30 NoAction:      NOP
31
32
33 ; Вместо NOP, разумеется, должны быть какие-нибудь полезные команды.
```

Сложно? Вот и я думаю, что делать нефиг :))))) Ничуть не сложней чем на Си забабахать из if then конструкцию или switch-case какой-нибудь.

Бег по граблям

У команд BR*** есть правда весьма ощутимое западло. Дальнобойность их составляет всего 63 команды. Т.е. это относительный переход. Поначалу ты этого не заметишь — короткая программа обычно не допускает переходов дальше 63.

Но вот, со временем, твоя программа распухнет, появится дофига кода и дальности бранча перестанет хватать. Вроде бы все работало, а добавил пару команд — и компилятор начал ругаться злыми словами — Error out of range или что то в этом духе. Как быть?

Перехватываться через промежуточные переходы. Т.е. находим ближайший к нам безусловный переход за который контроллер, как бы не старался залезть не сможет.

```
1  ActionA:        NOP
2                  NOP
3                  NOP
4                  RJMP    NoAction
5
6                  NOP             ; То есть если я вот тут поставлю островок
7                                  ; то он никогда не выполнится. Т.к. сверху
8                                  ; Процессор перепрыгнет сразу по RJMP
9                                  ; А снизу вход будет сразу же на ActionB
10
11 ActionB:        NOP
12                 NOP
13                 NOP
14                 RJMP    NoAction
15
16
17 И вот, в этом островке, мы создаем капсулу телепортер такого вида:
18
19
20 ActionA:        NOP
21                 NOP
22                 NOP
23                 RJMP    NoAction
24
25 ;-----------------------------------
```

```
26 Near:           JMP     FarFar_away
27 ;-----------------------------------
28
29
30 ActionB:        NOP
31                 NOP
32                 NOP
33                 RJMP    NoAction
```

Т.е. бранчу теперь достаточно дострелить до островка Near, а там дальнобойный JMP зашлет его хоть в бутсектор на другой конец флеша.

В случае большой программы, чтобы не плодить островки, его лучше сделать один, сразу после пачки CP.

Test & Skip
Кроме Branch'ей есть еще один тип команд: проверка — пропуск.
Работает она по принципу "Проверяем условие, если справедливо — пропускаем следующую команду"
Запомнить просто, первая буква это Skip — пропуск. Вторая условие — C=Clear, т.е. 0 S=Set, т.е. 1.
Соответственно S**C пропуск если сброшено. S**S пропуск если установлено.

Команды SBRC/SBRS

Указываешь ей какой РОН, и какой номер бита в этом регистре надо проверить. И если условие верное, то следующая команда будет пропущена.

Пример:

```
1          SBRC    R16,3   ; Если бит 3 в регистре R16 = 0, то прыжок через команду, на NOP
2
3          RJMP    bit_3_of_R16_Not_Zer0
4
5          NOP
6
7
8          SBRS    R16,3   ; Если бит 3 в регистре R16 = 1, то прыжок через команду, на NOP
9
10         RJMP    bit_3_of_R16_Zer0
11
12         NOP
```

Аналогичная команда

SBIC/SBIS Но проверяет она не биты регистров РОН, а биты регистров периферийных устройств. Те самые,что идут в памяти между блоком РОН и оперативкой. Но есть у ней ограничение — она может проверить только первые 1F регистров ввода вывода. Если заглянешь в m16def.inc (для мега16, для других МК в их файл дефайнов)

То увидишь там это:

```
1  .equ    UBRRH   = 0x20
2  .equ    UCSRC   = 0x20
3  .equ    EEARL   = 0x1e <-- а выше уже хрен :(
4  .equ    EEARH   = 0x1f <-- до сих можно обращаться через эти команды
5  .equ    EEDR    = 0x1d <-- ну и до нуля вниз.
6  .equ    EECR    = 0x1c
7  .equ    PORTA   = 0x1b
8  .equ    DDRA    = 0x1a
9  .equ    PINA    = 0x19
10 .equ    PORTB   = 0x18
11 .equ    DDRB    = 0x17
12 .equ    PINB    = 0x16
13 .equ    PORTC   = 0x15
```

```
14 .equ    DDRC    = 0x14
15 .equ    PINC    = 0x13
16 .equ    PORTD   = 0x12
17 .equ    DDRD    = 0x11
18 .equ    PIND    = 0x10
19 .equ    SPDR    = 0x0f
20 .equ    SPSR    = 0x0e
21 .equ    SPCR    = 0x0d
22 .equ    UDR     = 0x0c
23 .equ    UCSRA   = 0x0b
24 .equ    UCSRB   = 0x0a
25 .equ    UBRRL   = 0x09
26 .equ    ACSR    = 0x08
27 .equ    ADMUX   = 0x07
28 .equ    ADCSRA  = 0x06
29 .equ    ADCH    = 0x05
30 .equ    ADCL    = 0x04
31 .equ    TWDR    = 0x03
32 .equ    TWAR    = 0x02
33 .equ    TWSR    = 0x01
34 .equ    TWBR    = 0x00
```

Вот все это богатство твое. Проверяй не хочу :))))

А мне мало! Я хочу дальше!!! Что делать???
А дальше жизнь наша трудна и опасна. Чуть вправо,чуть влево — взрыв и ошметки. Страшно?

На самом деле ничего такого. Просто нам придется заюзать битмаски. И будет это не в одну команду, а в три.

Например, надо нам проверить состояние третьего бита в регистре UCSRC. Как видишь, это выше чем 1F Так что тест-скип не прокатит. Не беда, делай как я!

```
1         IN      R16, UCSRC      ; Хватаем байт из регистра, целиком.
2         ANDI    R16,1<<3        ; Накладываем на него маску 00001000
3                                 ; В результате, все байты кроме третьего
4                                 ; Обнулятся. А третий каким был таким и
5                                 ; останется. Был нулем - будет нуль и будет Z
6                                 ; Не был нулем, не будет Z флага. =))))
7
8         BREQ    Bit_3_of_R16_Equal_Zero
```

Говорил же в три команды уложимся =)

А можно проверять и не по одному биту сразу. Накладывай нужные маски, оставляй только нужные биты. А дальше смотри какие числа эти биты могут образовать и сравнивай напрямую с этими числами через CPI. Красота!!!

Ладно, к битмаскам мы еще не раз вернемся. Это мощнейшее средство, но пока, не влезая в периферию, его не прочуешь. А до периферии мы еще не до росли. Всему свое время.

В следующей статье я тебе буду взрывать мозг индексными табличными переходами. Сходи лучше воздухом подыши предварительно. Такие вещи надо раскуривать со свежими мозгами

AVR. Учебный курс. Ветвления на индексных переходах

Таблицы переходов
Вот представь, что нам надо сделать мега CASE, когда на вход валится число от 1 до 100 и нам надо сто вариантов действий.

Как будешь делать? Если лепить сто штук CPI с последующими переходами, то можно дальше убиться головой об стену. У тебя только эти CPI/BR** сожрут половину памяти кристалла. Учитывая, что каждая CPI это два байта, а каждый BR** еще байт. А о том сколько тактов эта шняга будет выполняться я даже не упоминаю.

Делается это все круче. Помнишь я тебе рассказывал в прошлых уроках о таких командах как ICALL и IJMP. Нет, это не новомодная яблочная истерия, а индексный переход. Прикол в том, что переход (или вызов подпрограммы, не важно) осуществляется тут не по метке, а по адресу в регистре Z (о том что Z это пара R30:R31 я пожалуй больше напоминать не буду, пора бы запомнить).

Итак, вначале пишем дофига вариантов наших действий. Те самые сто путей, у меня будет не сотня, а всего пять, но это не важно.

```
1 Way0:   NOP
2 Way1:   NOP
3 Way2:   NOP
4 Way3:   NOP
5 Way4:   NOP
```

Дальше, где нибудь посреди кода мы херачим таблицу переходов.

```
1 Table:  .dw     Way0, Way1, Way2, Way3, Way4
```

Распологать ее можно где угодно. Хоть прямо тутже, хоть в конце кода. Главное, чтобы программа при исполнении не выполнила эту строку как команды, иначе будет ошибка. Для этого таблицы либо перепрыгивают с помощью команды RJMP

```
1                 RJMP    Kudato
2 ; Программа сюда никогда не попадет.
3 Table:  .dw     Way0, Way1, Way2, Way3, Way4
4
5
6 Kudato:         NOP
7                 NOP
```

Или (предпочтительней) размещать в слепых тупиках кода, где выполнение не будет по алгоритму. Например между подпрограммами. Так:

```
1                  ...
2                  NOP
3                  RET     ; Точка выхода
4
5  ; Программа сюда никогда не попадет
6  Table:  .dw     Way0, Way1, Way2, Way3, Way4
7
8  SomeProc:       NOP     ; Точка входа
9                  NOP
10                 ...
```

Таблица переходов это всего лишь строка данных в памяти, содержащая адреса Way0…Way4. Обратите внимание на то, что данные у нас двубайтные слова dw!!! Если же мы хотим адресовать расположенные данные, например вложенную таблицу адресов переходов, то адреса нужно умножать на два.

Вообще тут проще при написании по быстрому скомпилить кусок, открыть дамп с map файлом и посмотреть что лежит в памяти, куда что ссылается. Сразу станет понятно надо умножать на два или нет.

Допустим у нас данные появляются в регистре R20 и нам, на основании числа там, нужно выбрать по какому пути переходить. Регистр R21 временный, под всякую хрень.

А теперь делаем финт ушами.

```
          LSL     R20             ; Сдвигом влево умножаем содержимое R20 на два.
                                  ; Было, например, 011=3  стало 110=6 Круто да? ;)
                                  ; опять же изза того, что у нас адреса двубайтные
1                                 ; НЕ ПУТАТЬ С ТЕМ ЧТО КОМПИЛЕР ОБСЧИТЫВАЕТ
2                                 ; ПАМЯТЬ В СЛОВАХ, тут несколько иное. Если непонятно
3                                 ; то в комменты пишите, обьясню.
4
5         LDI     ZL, low(Table*2)       ; Загружаем адрес нашей таблицы. Компилятор сам
6  посчитает
7         LDI     ZH, High(Table*2)      ; Умножение и запишет уже результат.Старший и
8  младший байты.
9
10        CLR     R21             ; Сбрасываем регистр R21 - нам нужен ноль.
11        ADD     ZL, R20         ; Складываем младший байт адреса. Если возникнет
12 переполнение
13        ADC     ZH, R21         ; То вылезет флаг переноса. Вторая команда складывает
14                                ; с учетом переноса. Поскольку у нас R21=0, то по сути
15                                ; мы прибавляем только флаг переноса. Вопросы - в
16 комменты!
17                                ; Таким образом складываются многобайтные числа.
18                                ; Позже освещу тему
19                                ; Математики на ассемблере.
20
21        LPM     R20,Z+          ; Загрузили в R20 адрес из таблицы
22        LPM     R21,Z           ; Старший и младший байт
23
24
25        MOVW     ZH:ZL,r21:r20          ; забросили адрес в Z
26
27
28 /* Что это было? А все просто! У нас наше число образовало смещение по таблице
29 переходов. Т.е. когда оно равно 0, то смещение тоже нулевое равное адресу Table,
30 а значит выбирается адрес ячейки где лежит адрес на Way0. Следом за ней в памяти сразу
31 же
32 находится адрес Way1, Разница между ними два байта (так как сами адреса двубайтные).
33 Таким образом, если в R20 будет число 1, то команда LSL умножит его на 2 и будет указан
34 на
35 адрес ячейки с Way1 и так далее.  А что же дальше? А дальше, конечно же: */

          IJMP                    ; Обана и переход на наш выбранный Way. Понравилось?
                                  ; То ли еще будет :)
```

А если хотите повзрывать себе мозг, то пофтыкайте в исходник [1] который я писал для статьи про трояна в мобильный телефон на базе Атмеги. Там конвейер за конвейером. Текстовые строки превращаются в последовательности АТ команд, натыкивающих буквы на сотовом телефоне (Б, например, это два нажатия на двойку и так далее). И вся это бодяга в виде кучи адресных таблиц хранится в памяти. Короче, укур еще тот

Камрады тут напоминают, что данный метод может быть реализован далеко не во всех AVR. Да это так, дело в том, что система команд у разных AVR немного отличается, например некоторые младшие модели (вроде Tiny 11/12) не могут делать индексные переходы и вызовы. Так что тут надо смотреть внимательно систему команд на конкретный контроллер. Но обычно такая фича есть.

Также, можно объединять конструкцию из CPI/BREQ с таблицами переходов. Чтобы получать неравномерные таблицы, в которых значения идут не по порядку. Т.е. мы сначала вычленяем в коде закономерности, которые можно увязать в таблицы, а все что выпадает в виде исключений обрабатываем отдельными CPI/BR** конструкциями.

В результате получаем очень компактный и быстрый код. Который не сможет переплюнуть по оптимальности ни один оптимизирующий компилятор.

Командные конвейеры
Еще мощнейшим инструментом является конструкция под названием командный конвейер. Термин я сам придумал ;), так что не придирайтесь.

Идея в чем — у нас есть множество процедур, делающий какие либо односложные операции. Если, например, это ЧПУ станок то операции могут быть такие:

- Поднять резец
- Опустить резец
- Включить подачу
- Включить привод
- Подать на один шаг вперед
- Подать на один шаг назад
- Подать влево
- Подать вправо.

А нам надо выточить деталь. И для этого есть ТЗ, где сказано в какой последовательности эти операции должны быть выполнены, чтобы получить нужную деталь. Причем программа то у нас одна, а последовательности разные.

Не переписывать же код каждый раз. В этом случае можно сделать командный конвейер.

Команды у нас четко определены, так что мы их можем записать в виде процедур ассемблерных:

```
1  BladeUP:        NOP
2                  NOP
3                  NOP
4                  NOP
5                  RET
6
7  BladeDN:        NOP
8                  NOP
9                  NOP
10                 NOP
11                 RET
12
13 DriveON:        NOP
14                 NOP
15                 NOP
16                 NOP
17                 RET
18
19 DriveOFF:       NOP
20                 NOP
21                 NOP
22                 NOP
23                 RET
24
25 Forward:        NOP
26                 NOP
27                 NOP
28                 NOP
29                 RET
30
31 Back:           NOP
32                 NOP
33                 NOP
34                 NOP
35                 RET
```

И так далее, все нужные команды.

Затем, также как и в случае индексных переходов, создается таблица с адресами процедур.

```
1 Index:          .dw     BladeUP, BladeDN, DriveON, DriveOFF, Forward, Back
```

А в памяти ОЗУ заводится очередь задач. Обычный строковый массив нужной длины:

```
1 TaskList:       .byte   30
```

Потом создается процедура которая извне загружает нашу очередь действий, в виде последовательности кодов этих действий (всего лишь смещение по таблице! Очень быстрое и компактное!) и вызывается диспетчер очереди.

Диспетчер будет в цикле перебирать нашу строку TaskList, выковыривать из нее номера подпрограмм, по таблице, смещением, вычислять адреса переходов на реальные процедуры, переходить по ним, делать полезное действие, возвращаться обратно и брать новое значение из таблицы.

Чуете куда я клоню? Получается своеобразная виртуальная машина. Процессор в процессоре. Более того, сами задачи, вызываемые из очереди, не обязательно должны быть тупо выполняющими конкретное действие. Они же тоже могут набрасывать в эту самую очередь новых кодов, сортировать и на ходу их переделывать. Получается мощный полиморфный алгоритм, который может сам менять свою логику исходя из условий.

Позже, я на этом принципе покажу прообраз операционной системы. С диспетчером задач и кучей потоков, вызывающих друг друга по цепочке.

А еще на индексных переходах можно делать очень компактные конечные автоматы. Причем даже адрес не надо будет вычислять т.к. в виде текущего состояния можно смело использовать прямые адреса других состояний автомата.

Если не понял о чем идет речь, то не забивай пока голову. Дальше будет на это дело отдельная статья =) Или покури интернет на предмет того, что такое конечные автоматы и как они реализуются.

AVR. Учебный Курс. Типовые конструкции

При написании программ постоянно приходится использовать разные стандартные конструкции вроде циклов, операторов выбора, перехода, сравнения. Всякие там if-then-else или case-switch. В высокоуровневых языках это все уже готово, а на ассемблере приходится изобретать каждый раз заново.
Впрочем, такие вещи пишутся один раз, а дальше просто по наезженной тропинке применяются везде, где потребуется. Заодно и оптимизируются по ходу процесса.

Условие if-then-else
Тут проще всего методом последовательной проверки проложить маршрут до нужного блока кода. Приведу пример:

```
1 if (A>=B)
2         {
3         action_a
4         }
5 else
6         {
7         action_b
8         }
9 next_action
```

Как это проще всего сделать на ассемблере?

Считаем, что A в R16,B в R17, а полезные действия, которые могли бы быть там, заменяем на NOP,NOP,NOP. Сравнение делается путем вычитания. По итогам вычитания выставляются флаги. Заем (когда A стало меньше B) означает, что условие не выполнилось.

```
1                 CP      R16,R17         ; Сравниваем два значения
2                 BRCS    action_b        ; когда A>=B флага C не будет
3                                         ; Перехода не произойдет
```

```
4  action_a:       NOP                     ; и выполнится действие A
5                  NOP
6                  NOP
7                  RJMP next_action        ; Но чтобы действие B не произошло
8                                          ; в ходе естественного выполнения
9                                          ; кода -- его надо перепрыгнуть.
10
11 action_b:       NOP                     ; Действие B
12                 NOP
13                 NOP
14
15
16 next_action:    NOP
17                 NOP
18                 NOP
```

Но надо учитывать тот момент, что в случае A=B флаг C не вылезет, зато будет флаг Z. Но переход то у нас исходя из какого-нибудь одного флага (по C=0 или по C=1). И, исходя из выбора команды для построения конструкции if-then-else (BRCC или BRCS), будет разная трактовка результата условия if (A>=B) в случае A=B.

В одном случае (нашем), где переход идет на else по C=1, A=B будет эквивалентно A>B — флага C не будет. И в том и другом случае услове if (A>=B) даст True и переход по BRCS на then.

Если перестроить конструкцию наизнанку, через команду BRCC то условия (A>=B) уже не получится. При A=B не будет флага C и произойдет переход по BRCC на else.

Для создания строгих неравенств нужна проверка на ноль.
Теперь A больше B:

```
1 if (A>B)
2         {
3         action_a
4         }
5 else
6         {
7         action_b
8         }
9 next_action
```

Получили:

```
1                  CP      R16,R17         ; Сравниваем два значения
2                  BREQ    action_b        ; Если равно (флаг Z), то переход сразу.
3                                          ; Потом проверяем второе условие.
4                  BRCS    action_b        ; когда A>B флага C не будет
5                                          ; Перехода не произойдет
6
7  action_a:       NOP                     ; и выполнится действие A
8                  NOP
9                  NOP
10                 RJMP next_action        ; Но чтобы действие B не произошло
11                                         ; в ходе естественного выполнения
12                                         ; кода -- его надо перепрыгнуть.
13
14 action_b:       NOP                     ; Действие B
15                 NOP
16                 NOP
17
18
19 next_action:    NOP
20                 NOP
```

```
21                NOP
```

Сложно? Думаю нет.

Усложним еще раз наше условие:

```
1 if (C<A AND A<B)
2         {
3         action_a
4         }
5 else
6         {
7         action_b
8         }
9 next_action
```

Считаем, что A=R16, B=R17, C=R18

```
   ;IF
1                 CP      R16,R18         ; Сравниваем C и A
2                 BREQ    action_b        ; Переход сразу на ELSE если C=A
3                 BRCS    action_b        ; Если A оказалось меньше, то будет C
4                                         ; и сразу выходим отсюда на else
5                                         ; Но если C<A то идем дальше и проверяем
6                                         ; Второе условие (A<B)
7
8                 CP      R16,R17         ; Сравниваем два значения
9                 ; BREQ  action_b        ; Переход на ELSE если B=A. Команда BREQ
10 специально
11                                        ; закомментирована -- она тут не нужна.  Я ее
12 поставил
13                                        ; для наглядности. Ведь в случае A=B
14                                        ; флага C не будет и однозначно будет переход по
15 BRCC.
16
17                BRCC    action_b        ; Когда A>B флага C не будет
18                                        ; А переход по условию C clear сработает.
19 ;THEN
20 action_a:      NOP                     ; Выполнится действие A
21                NOP
22                NOP
23                RJMP next_action        ; Но чтобы действие B не произошло
24                                        ; в ходе естественного выполнения
25                                        ; кода -- его надо перепрыгнуть.
26 ;ELSE
27 action_b:      NOP                     ; Действие B
28                NOP
29                NOP
30
31
32 next_action:   NOP
                  NOP
                  NOP
```

Просто же! Вот так вот, комбинируя условия, можно размотать любую логическую конструкцию.

Битовые операции
Но, пожалуй, самые распространенные операции в контроллере — битовые. Включить выключить какой-нибудь параметр, инвертировать его, проверить есть или нет. Да масса случаев где это нужно. Основная масса операций идет через битовые маски.

Есть замечательные команды SBI и CBI первая ставит указанный бит в порту, вторая сбрасывает. Например:

CBI PORT,7 — обнулить 7й бит в регистре ввода-вывода PORT
SBI PORT,6 — выставить 6й бит в регистре ввода-вывода PORT

Все замечательно, но работают эти команды только в пределах первых 31 адресов в пространстве регистров ввода вывода. Для мега16 это:

```
1  .equ    EEARH   = 0x1f
2  .equ    EEDR    = 0x1d
3  .equ    EECR    = 0x1c
4  .equ    PORTA   = 0x1b
5  .equ    DDRA    = 0x1a
6  .equ    PINA    = 0x19
7  .equ    PORTB   = 0x18
8  .equ    DDRB    = 0x17
9  .equ    PINB    = 0x16
10 .equ    PORTC   = 0x15
11 .equ    DDRC    = 0x14
12 .equ    PINC    = 0x13
13 .equ    PORTD   = 0x12
14 .equ    DDRD    = 0x11
15 .equ    PIND    = 0x10
16 .equ    SPDR    = 0x0f
17 .equ    SPSR    = 0x0e
18 .equ    SPCR    = 0x0d
19 .equ    UDR     = 0x0c
20 .equ    UCSRA   = 0x0b
21 .equ    UCSRB   = 0x0a
22 .equ    UBRRL   = 0x09
23 .equ    ACSR    = 0x08
24 .equ    ADMUX   = 0x07
25 .equ    ADCSRA  = 0x06
26 .equ    ADCH    = 0x05
27 .equ    ADCL    = 0x04
28 .equ    TWDR    = 0x03
29 .equ    TWAR    = 0x02
30 .equ    TWSR    = 0x01
31 .equ    TWBR    = 0x00
```

И ни байтом дальше. А с более старшими адресами — облом.

Ну ничего, что нам мешает взять и записать в регистр ввода-вывода (далее РВВ, а то я уже задолбался). Этот бит самому?

Только то, что доступ там к порту идет не побитный, а сразу целыми байтами.
Казалось бы, в чем проблема? Надо выставить биты 1 и 3 в регистре TWCR, например, взял да записал двоичное число 00001010 и выставил.

```
1         LDI     R16,1<<3|1<<1
```

Да, можно и так, но если при этом другие биты в этом регистре равны нулю, а то ведь мы их все накроем сразу.

В таком случае, алгоритм у нас такой:

- Взять старое значение
- Подправить в нем биты
- Записать обратно

Запишем это в коде. Взять значение просто:

```
1         IN      R16,TWCR
```

А установить нужные биты нам поможет битовая маска и операция OR. В результате этой операции, там где в битовой маске были нули будут те же значения, что и раньше. А где в маске 1 возникнут единички.

```
1         ORI     R16,1<<3|1<<1   ; Битовая маска 00001010
```

А затем записать уже измененный байт обратно

```
1         OUT     TWCR,R16
```

Сброс битов тоже можно сделать через битовую маску, но уже операция AND. И там где мы хотим сбросить нам надо поставить в маске 0, а где не хотим менять 1

Удобней делать инверсную маску. Для инверсии маски применяется операция побитового НЕ ~

Выглядеть будет так:

```
1         IN      R16,TWCR
2         ANDI    R16,~(1<<3|1<<1)
3         OUT     TWCR,R16
```

Обрати внимание, для сброса у нас внутри конструкции используется 1. Ведь байт то потом инвертируется!

Для инверсии битов применяется маска по XOR. Нули в этой маске не меняют биты, а единичка по XOR бит инвертирует.

Смотри сам, вверху произвольное число, внизу маска:

```
1 1100 1100 XOR
2 1111 0000
3 _________
4 0011 1100
```

Правда есть одно ограничение — нет команды XOR маски по числу, поэтому через регистр.

```
1         IN      R17,TWCR
2         LDI     R16, 3<<1|2<<1  ; Маска
3         EOR     R17,R16                 ; Ксорим
4         OUT     TWCR,R17                ; Сгружаем обратно
```

Можно увязать всю ботву в макросы и тогда будет совсем хорошо :) Единственно, при использовании макросов, особенно когда они содержат внутри промежуточные регистры, нельзя забывать про эти регистры. А то в ходе исполнения макроса содержимое этих регистров меняется и если про это забыть можно получить кучу глюков :)

Но от такой напасти можно и защититься, например стеком. К примеру, изобретем мы свою команду XRI — XOR регистра с числом.

Макрос может быть таким:

```
1         .MACRO  XRI
2         LDI     @2,@1
3         EOR     @0,@2
4         .ENDM
```

А вызов макроса

```
1         XRI     R16,mask,R17
```

Последний параметр — промежуточный регистр. Указываем его вручную т.к. мало ли какой из регистров нам будет удобней в данный момент.

А можно сделать с прогрузом стека, тогда промежуточные регистры вообще не нужны. Но будет выполняться дольше, занимать больше памяти и требовать стек.

```
1         .MACRO  XRI
2         PUSH    R16
3         LDI     R16,@1
4         EOR     @0,R16
5         POP     R16
6         .ENDM
```

Вызов, в этом случае

```
1         XRI     R17,Mask
```

Но нельзя будет поксорить R16, т.к. его значение все равно затрет при выгрузке из стека.

Также масками можно смело отрезать неиспользованные биты. Например, в регистре TWSR первые два бита это настройки, а остальные — код состояние интерфейса TWI. Так зачем нам эти два бита настройки каждый раз учитывать? Отдавили их по AND через маску 11111100 да дело в шляпе. Или сдвинули вправо.

Сдвиги
Ну тут просто — можно двигать биты в РОН влево и вправо. Причем сдвиг бывает двух типов.

LSR Rn — логический вправо, при этом слева в байт лезут нули.
LSL Rn — логический влево, при этом справа в байт лезут нули.

и через перенос.

ROL и ROR

При этом байт уходящий за край попадает в флаг С, а флаг С вылезает с другого края байта. Получается как бы 9 битная циклическая карусель. Где одна команда дает один шаг влево или вправо.

Как применить? Ну способы разные, например, посчитать число единиц в байте.

Смотри как просто:

```
1          CLR     R18             ; Сюда будем считать единички. Сбросим пока
2          LDI     R17,9           ; Счетчиком циклов будет
3          LDI     R16,0xAA        ; В этом байте будем считать единички
4          CLC                     ; Сбросим флаг С, чтобы не мешался.
5
6  Loop:   DEC     R17             ; уменьшим счетчик
7          BREQ    End             ; если весь байт дотикали - выход.
8
9          ROL     R16             ; Сдвигаем байт
10         BRCC    Loop            ; если нет 1 в С, еще одну итерацию
11
12         INC     R18             ; Сосчтитали единичку
13         RJMP    Loop
14
15 End:    NOP
```

Еще через флаг C удобно проверять крайние биты, по быстрому, чтобы не заморачиваться с масками (хотя выигрыша ни по времени, ни по коду не будет).

```
1         ROL     R16
2         BRCS    Label   ; Переход если бит 7 в R16 есть.
```

или

```
1         ROR     R16
2         BRCS    Laber   ; Переход если бит 0 в R16 есть.
```

Циклы
Ну тут все просто — либо сначала, либо потом условие, а затем тело цикла и все окольцовано переходами.

Например, выполнить цикл 20 раз.

```
1          LDI     R17,20          ; Счетный регистр
2
3  Loop:   NOP
4          NOP
5          NOP
6          NOP
7          NOP
8
9          DEC     R17             ; Уменьшаем счетчик
10         BRNE    Loop
```

; Переход если не ноль (нет флага Z)

Главное следить за тем, чтобы счетный регистр нигде не запоролся в теле цикла. Иначе получим глюк. Такой глюк, конечно, легко вылавливается трассировкой, но далеко не факт, что он всплывает сразу.

Еще один пример цикла — тупые задержки. Почему я их называю тупыми? Да потому, что они сами ничего не делают и другим не дают. Крайне не рекомендую их использовать, но для простеньких программ подойдет.

Основная идея такой задержки загрузить контроллер бессмысленной работой, чтобы нащелкать как можно больше тактов. У каждого такта есть своя длительность, так что длительность задержки можно формировать от микросекунд, до десятков лет.

Вот решение в лоб:

```
1         LDI     R16, Delay
2 Loop:   DEC     R16
3         BRNE    loop
```

Процессор потратит примерно Delay*2 тактов на бессмысленные операции. При длительности такта (на 8мгц) 1.25E-7 секунд максимальная выдержка будет 6.4E-5 секунды. Немного, но, например, хватит на то чтобы дисплей прожевал отданый ему байт и не подавился следующим.

Если надо больше задержки, то делаются вложенные циклы с несколькими счетчиками. Тогда суммарная длительность выдержки перемножается и уже на трех вложенных циклах можно получить около 2 секунд. Ну, а на четырех уже 536 секунд.

Но есть более красивое решение, нежели вложенные циклы — вычитание многобайтного числа с переносом.

```
1         LDI     R16,LowByte             ; Грузим три байта
2         LDI     R17,MidleByte           ; Нашей выдержки
3         LDI     R18,HighByte
4
```

```
5 loop:   SUBI    R16,1                 ; Вычитаем 1
6         SBCI    R17,0                 ; Вычитаем только C
7         SBCI    R18,0                 ; Вычитаем только C
8
9         BRCC    Loop                  ; Если нет переноса - переход.
```

В результате, вначале у нас отнимается 1 из первого числа, а перенос возникает только тогда, когда заканчивается очередной байт. Получается, что из R16 отнимают на каждой итерации, из R17 на каждой 256 итерации, а из R18 на каждой 65535 итерации.

А всего на таких трех регистрах можно замутить тупняк на 256*256*256 тактов. И никто не мешает в том же ключе навешать еще регистров, вплоть до R31 =) И длительность задержки будет куда точней, т.к. в отличии от вложенных циклов, команда BRCC всегда будет выполняться за 2 такта и лишь в последней случае за один.

Но, опять же, повторюсь, что данные задержки это отстой. Если нужна большая точность — применяют таймеры. Если длительные выдержки, то программные таймеры и цикловые счетчики. О таймерах я расскажу чуть поздней, в периферии, а вот на цикловом счетчике можно остановиться чуть подробней.

Цикловой счетчик
Если нам надо просто затупить, но точность не нужна совершенно — плюс минус несколько миллисекунд не решают. Например, когда задержка нужна лишь для того, чтобы увидеть, что что то происходит, затормозить процесс.

Как у нас в случае тупой задержки выглядит алгоритм:

- Делаем раз
- Делаем два
- Делаем три
- Тупим
- Делаем то, ради чего тупили
- Делаем четыре

Как понимаешь, пока длится этап «Тупим» ничего не работает. Если в «делай раз» было, например, обновление экрана, то оно зависнет. На период этапа «Тупим»

Для этого можно сделать финт ушами — полезную работу ВСЕЙ программы внести в цикл задержки.

Получается так:

- Делай раз
- Делай два
- Делай три
- Тик!
- Натикало? -да- Делаем то, ради чего тупили сбрасываем счетчик.
- Делай четыре

Покажу пример (инициализации стека, и таблицы векторов я опускаю для краткости):

```
1          .DSEG
2  Counter:        .byte   3                    ; Наш счетчик циклов. Три байта.
3
4                  .CSEG
5
6  MainLoop:
7
8  Do_One:         NOP
9
10 Do_Two:         NOP
11
12 Do_Three:       NOP
```

```
13
14                LDS     R16,Counter           ; Грузим наш счетчик
15                LDS     R17,Counter+1
16                LDS     R18,Counter+2
17
18                SUBI    R16,1                 ; Вычитаем 1
19                SBCI    R17,0                 ; Вычитаем только C
20                SBCI    R18,0                 ; Вычитаем только C
21
22                BRCC    DoNothing             ; Не натикало? Переход
23
24 ; Натикало!
25 YES:           NOP                           ; Делаем то, ради чего тупили
26                NOP
27                NOP
28
29                LDI     R16,LowByte           ; Грузим три байта
30                LDI     R17,MidleByte         ; Нашей выдержки
31                LDI     R18,HighByte
32
33
34 DoNothing:     STS     Counter,R16           ; Сохраняем обратно в память
35                STS     Counter+1,R17
36                STS     Counter+2,R18
37
38                RJMP    MainLoop
```

Обрати внимание, что другие операции Do_one, Do_Two, Do_Three не ждут когда натикают, они выполняются в каждую итерацию. А операция требующая задержку ее получает, не тормозя всю программу! И таких таймеров можно налепить сколько угодно, пока оперативка не кончится.

Более того, такой цикловой счетчик сам может быть алгоритмозадающим механизмом. Счетчик тикает каждый прогон главного цикла, а каждая операция в этом главном цикле ждет именно своего «номера» в главном счетчике. Получается этакая программная «шарманка», где в качестве барабана с гвоздями выступает наш счетчик.

Но это уже конечные автоматы. К ним я, возможно, вернусь позже. Еще не решил стоит освещать эту тему. А то могу и увлечься. люблю я их :)

Горыныч
Писал я тут библиотечку для подключения LCD к AVR. А чтобы не вкуривать в команды контроллера дисплея я распотрошил Сишный код, дабы подглядеть какими байтами и в каком порядке надо кормить контроллер, чтобы он вышел на нужный режим.

Попутно сварганил один прикольный трюк, который я называю Горыныч. Это когда мы несколько функций объединяем в одну многоголовую — с несколькими точками входа и одной точкой выхода.

Итак, мы имеем две функции обращения к LCD — запись данных и запись команд.

О! Вот эти ребята:

```
1  CMD_WR:        CLI
2                 RCALL   BusyWait
3
4                 CBI     CMD_PORT,RS
5                 CBI     CMD_PORT,RW
6                 SBI     CMD_PORT,E
7                 LCD_PORT_OUT
8                 OUT     DATA_PORT,R17
9                 RCALL   LCD_Delay
10                CBI     CMD_PORT,E
11                LCD_PORT_IN
```

```
12                SEI
13                RET
```

Нененененене Девид Блейн!!!!

```
1  DATA_WR:       CLI
2                 RCALL   BusyWait
3
4                 SBI     CMD_PORT,RS
5                 CBI     CMD_PORT,RW
6                 SBI     CMD_PORT,E
7                 LCD_PORT_OUT
8                 OUT     DATA_PORT,R17
9                 RCALL   LCD_Delay
10                CBI     CMD_PORT,E
11                LCD_PORT_IN
12                SEI
13                RET
```

Сейчас я вам покажу особую, оптимизаторскую, магию!

```
1  CMD_WR:        CLI
2                 RCALL   BusyWait
3
4                 CBI     CMD_PORT,RS
5                 RJMP    WR_END
6
7  DATA_WR:       CLI
8                 RCALL   BusyWait
9                 SBI     CMD_PORT,RS
10 WR_END: CBI    CMD_PORT,RW
11                SBI     CMD_PORT,E
12                LCD_PORT_OUT
13                OUT     DATA_PORT,R17
14                RCALL   LCD_Delay
15                CBI     CMD_PORT,E
16                LCD_PORT_IN
17                SEI
18                RET
```

Ты что наделал Блейн!!! Ты зачем функцию скукожил??? Нука раскукожь ее обратно! У меня теперь функция беби сайз!!!

Ну, а теперь, для сравнения, поглядим как это сделано в **LCD.c** весь исходник я сюда копировать не буду, только то, что сделано у меня, ну и всю условную компиляцию я тоже выкину.

```
1  void lcdControlWrite(u08 data)
2  {
3  lcdBusyWait();
4  cbi(LCD_CTRL_PORT, LCD_CTRL_RS);
5  cbi(LCD_CTRL_PORT, LCD_CTRL_RW);
6
7  sbi(LCD_CTRL_PORT, LCD_CTRL_E);
8  outb(LCD_DATA_DDR, 0xFF);
9  outb(LCD_DATA_POUT, data);
10 LCD_DELAY;
11 LCD_DELAY;
12 cbi(LCD_CTRL_PORT, LCD_CTRL_E);
13
14 outb(LCD_DATA_DDR, 0x00);
15 outb(LCD_DATA_POUT, 0xFF);
```

```
16 }
17
18
19 void lcdDataWrite(u08 data)
20 {
21 lcdBusyWait();
22 sbi(LCD_CTRL_PORT, LCD_CTRL_RS);
23 cbi(LCD_CTRL_PORT, LCD_CTRL_RW);
24 sbi(LCD_CTRL_PORT, LCD_CTRL_E);
25 outb(LCD_DATA_DDR, 0xFF);
26 outb(LCD_DATA_POUT, data);
27 LCD_DELAY;
28 LCD_DELAY;
29 cbi(LCD_CTRL_PORT, LCD_CTRL_E);
30 outb(LCD_DATA_DDR, 0x00);
31 outb(LCD_DATA_POUT, 0xFF);
32 }
```

Собственно, строчка в строчку — те же яйца только в профиль.
Что подтверждает, сказанный мной в первом посте, тезис, что не важно на каком языке писать, на си или на ассемблере — главное знать что в какие порты пихать.

Но на асме я могу свернуть свою прогу почти вдвое, не потеряв ни байта и ни такта, а на сях придется для этого вводить внутреннюю функцию, которая захавает оперативку на стек, потребует кучу тактов на переходы и вообще будет выглядеть убого.

И финальная процедура инициализации на сях и на макроасме:

```
1  void lcdInit()
2  {
3          lcdInitHW();
4          lcdControlWrite(LCD_FUNCTION_DEFAULT);
5          lcdControlWrite(1<<LCD_CLR);
6          lcdControlWrite(1<<LCD_ENTRY_MODE | 1<<LCD_ENTRY_INC);
7          lcdControlWrite(1<<LCD_ON_CTRL | 1<<LCD_ON_DISPLAY );
8          lcdControlWrite(1<<LCD_HOME);
9          lcdControlWrite(1<<LCD_DDRAM | 0x00);
10 }
```

Ну и у меня:

```
1         .MACRO  INIT_LCD
2         RCALL   InitHW
3         WR_CMD  0x38
4         WR_CMD  0x01
5         WR_CMD  0x06
6         WR_CMD  0x0F
7         WR_CMD  0x02
8         .ENDM
```

Конструкции вида (1<<LCD_HOME) обусловены тем, что Сишный исходник подразумевает конфигурацию LCD — это универсализация кода.

Я тоже мог бы сбодяжить на макросах такое, но мне было лень. Поэтому я просто забил байты. По сути дела разницы нет. Правильно написанный ассемблерный код на очень многих задачах (особенно мелких, вроде этого LCD) куда более компактный и быстрый, **без снижения читабельности**.

Конечно что либо крупное, вроде того же USB я на Асме делать не буду, тут проще взять готовый код.

Хотя… чем черт не шутит, может заморочусь и напишу свой крошечный USB драйвер.

AVR. Учебный курс. Стартовая инициализация

Инициализация памяти
Мало кто подозревает о том, что при включении в оперативке далеко не всегда все байты равны 0xFF. Они могут, но не обязаны. Равно как и регистры РОН не всегда равны нулю при запуске. Обычно да, все обнулено, но я несколько раз сталкивался со случаями когда после перезапуска и/или включения-выключения питания, микроконтроллер начинал творить не пойми что. Особнно часто возникает когда питание выключаешь, а потом, спустя некоторое время, пара минут, не больше, включаешь. А всему виной остаточные значения в регистрах.

Итак, возьмите себе за правило после каждого включения, в разделе инициализации, еще даже до инициализации стека, делать зануление памяти и очистку всех регистров. Разумеется делается это все в цикле. Вот примерный вариант кода:

```
1  RAM_Flush:     LDI     ZL,Low(SRAM_START)     ; Адрес начала ОЗУ в индекс
2                 LDI     ZH,High(SRAM_START)
3                 CLR     R16                    ; Очищаем R16
4  Flush:         ST      Z+,R16                 ; Сохраняем 0 в ячейку памяти
5                 CPI     ZH,High(RAMEND+1)      ; Достигли конца оперативки?
6                 BRNE    Flush                  ; Нет? Крутимся дальше!
7
8                 CPI     ZL,Low(RAMEND+1)       ; А младший байт достиг конца?
9                 BRNE    Flush
10
11                CLR     ZL                     ; Очищаем индекс
12                CLR     ZH
```

Поскольку адрес оперативки у нас двубайтный, то мы вначале смотрим, чтобы старший байт совпал с концом, а потом добиваем оставшиеся 255 байт в младшем байте адреса.
Далее убиваем все регистры от первого до последнего. Все, контроллер готов к работе.

```
1                 LDI     ZL, 30          ; Адрес самого старшего регистра
2                 CLR     ZH              ; А тут у нас будет ноль
3                 DEC     ZL              ; Уменьшая адрес
4                 ST      Z, ZH           ; Записываем в регистр 0
5                 BRNE    PC-2            ; Пока не перебрали все не успокоились
```

За процедурку зануления регистров спасибо Testicq

Либо значения сразу же инициализируются нужными величинами. Но, обычно, я от нуля всегда пляшу. Поэтому зануляю все.

З.Ы.
Кстати, о оперативке. Нашел я недавно планку оперативной памяти на 1килобайт, древнюю как говно мамонта, еще на **ферромагнитных кольцах.** [1]

AVR. Учебный курс. Скелет программы

При написании прошивки надо очень внимательно подходить к процессу организации архитектуры будущей программы. Программа должна быть быстрой, не допускать задержек главного цикла и легко расширяться. Оптимально использовать аппаратные ресурсы и стараться выжать максимум возможного из имеющихся ресурсов.

Вообще, архитектура программ это отдельная тема и ближе к концу курса, в его Сишной части я подробней рассказываю о разных типах организации прошивки. Можешь забежать вперед и поглядеть, что да как.

В ассемблерной же части, я расскажу о одном из самых простых вариантов — флаговом автомате, а позже, когда ты уже будешь вовсю ориентироваться в моем коде, дам пример на основе конвейерного диспетчера, с подробным описанием его работы.

Суперцикл
Все программы на микроконтроллерах обычно зацикленные. Т.е. у нас есть какой то главный цикл, который вращается непрерывно.

Структура же программы при этом следующая:

- **Макросы и макроопредения**
- **Сегмент ОЗУ**
- **Точка входа** — ORG 0000
- **Таблица векторов** — и вектора, ведущие в секцию обработчиков прерываний
- **Обработчики прерываний** — тела обработчиков, возврат отсюда только по RETI
- **Инициализация памяти** — а вот уже отсюда начинается активная часть программы
- **Инициализация стека**
- **Инициализация внутренней периферии** — программирование и запуск в работу всяких таймеров, интерфейсов, выставление портов ввода-вывода в нужные уровни. Разрешение прерываний.
- **Инициализация внешней периферии** — инициализация дисплеев, внешней памяти, разных аппаратных примочек, что подключены к микроконтроллеру извне.
- **Запуск фоновых процессов** — процессы работающие непрерывно, вне зависимости от условий. Такие как сканирование клавиатуры, обновление экрана и так далее.
- **Главный цикл** — тут уже идет вся управляющая логика программы.
- Сегмент ЕЕПРОМ

Начинается все с макросов, их пока не много, если что по ходу добавим.

```
1         .include "m16def.inc"   ; Используем ATMega16
2
3  ;= Start macro.inc ======================================
4         .macro    OUTI
5         LDI    R16,@1
6         .if @0 < 0x40
7         OUT    @0,R16
8         .else
9         STS      @0,R16
10        .endif
11        .endm
12
13        .macro    UOUT
14        .if     @0 < 0x40
15        OUT     @0,@1
16        .else
17        STS     @0,@1
18        .endif
19        .endm
20 ;= End  macro.inc ====================================
```

В оперативке пока ничего не размечаем. Нечего.

```
1 ; RAM =============================================
2               .DSEG
3 ; END RAM =========================================
```

С точкой входа и таблицей векторов все понятно, следуя нашему давнему шаблону, берем его оттуда:

```
1  ; FLASH ============================================
2           .CSEG
```

```
3           .ORG $000      ; (RESET)
4           RJMP   Reset
5           .ORG $002
6           RETI             ; (INT0) External Interrupt Request 0
7           .ORG $004
8           RETI             ; (INT1) External Interrupt Request 1
9           .ORG $006
10          RETI         ; (TIMER2 COMP) Timer/Counter2 Compare Match
11          .ORG $008
12          RETI             ; (TIMER2 OVF) Timer/Counter2 Overflow
13          .ORG $00A
14          RETI       ; (TIMER1 CAPT) Timer/Counter1 Capture Event
15          .ORG $00C
16          RETI             ; (TIMER1 COMPA) Timer/Counter1 Compare Match A
17          .ORG $00E
18          RETI             ; (TIMER1 COMPB) Timer/Counter1 Compare Match B
19          .ORG $010
20          RETI             ; (TIMER1 OVF) Timer/Counter1 Overflow
21          .ORG $012
22          RETI             ; (TIMER0 OVF) Timer/Counter0 Overflow
23          .ORG $014
24          RETI             ; (SPI,STC) Serial Transfer Complete
25          .ORG $016
26          RETI               ; (USART,RXC) USART, Rx Complete
27          .ORG $018
28          RETI             ; (USART,UDRE) USART Data Register Empty
29          .ORG $01A
30          RETI             ; (USART,TXC) USART, Tx Complete
31          .ORG $01C
32          RETI       ; (ADC) ADC Conversion Complete
33          .ORG $01E
34          RETI             ; (EE_RDY) EEPROM Ready
35          .ORG $020
36          RETI             ; (ANA_COMP) Analog Comparator
37          .ORG $022
38          RETI             ; (TWI) 2-wire Serial Interface
39          .ORG $024
40          RETI             ; (INT2) External Interrupt Request 2
41          .ORG $026
42          RETI             ; (TIMER0 COMP) Timer/Counter0 Compare Match
43          .ORG $028
44          RETI             ; (SPM_RDY) Store Program Memory Ready
45
46          .ORG   INT_VECTORS_SIZE      ; Конец таблицы прерываний
```

Обработчики пока тоже пусты, но потом добавим

```
1 ; Interrupts ============================================
2 ; End Interrupts ========================================
```

Инициализация ядра. Память, стек, регистры:

```
1  Reset:          LDI R16,Low(RAMEND)           ; Инициализация стека
2                  OUT SPL,R16                   ; Обязательно!!!
3
4                  LDI R16,High(RAMEND)
5                  OUT SPH,R16
6
7  ; Start coreinit.inc
8  RAM_Flush:      LDI     ZL,Low(SRAM_START)    ; Адрес начала ОЗУ в индекс
9                  LDI     ZH,High(SRAM_START)
10                 CLR     R16                   ; Очищаем R16
```

```
11 Flush:          ST      Z+,R16                  ; Сохраняем 0 в ячейку памяти
12                 CPI     ZH,High(RAMEND)         ; Достигли конца оперативки?
13                 BRNE    Flush                   ; Нет? Крутимся дальше!
14
15                 CPI     ZL,Low(RAMEND)          ; А младший байт достиг конца?
16                 BRNE    Flush
17
18                 CLR     ZL                      ; Очищаем индекс
19                 CLR     ZH
20                 CLR     R0
21                 CLR     R1
22                 CLR     R2
23                 CLR     R3
24                 CLR     R4
25                 CLR     R5
26                 CLR     R6
27                 CLR     R7
28                 CLR     R8
29                 CLR     R9
30                 CLR     R10
31                 CLR     R11
32                 CLR     R12
33                 CLR     R13
34                 CLR     R14
35                 CLR     R15
36                 CLR     R16
37                 CLR     R17
38                 CLR     R18
39                 CLR     R19
40                 CLR     R20
41                 CLR     R21
42                 CLR     R22
43                 CLR     R23
44                 CLR     R24
45                 CLR     R25
46                 CLR     R26
47                 CLR     R27
48                 CLR     R28
49                 CLR     R29
50 ; End coreinit.inc
```

Всю эту портянку можно и нужно спрятать в inc файл и больше не трогать.

Секции внешней и внутренней инициализации переферии пока пусты, но ненадолго. Равно как и запуск фоновых программ. Потом я просто буду говорить, что мол добавьте эту ботву в секцию Internal Hardware Init и все :)

```
1  ; Internal Hardware Init  ====================================
2
3  ; End Internal Hardware Init ================================
4
5  ; External Hardware Init  ====================================
6
7  ; End Internal Hardware Init ================================
8
9  ; Run ========================================================
10
11 ; End Run ====================================================
```

А теперь, собственно, сам главный цикл.

```
1 ; Main =======================================================
2 Main:
```

```
3
4                   JMP     Main
5 ; End Main ======================================================
```

Все процедуры располагаются в отдельной секции, не смешиваясь с главным циклом. Так удобней, потом можно их по частям вынести в библиотечные файлы и разделить исходник на несколько файлов. Но мы пока это делать не будем. Разделим их просто логически.

```
1 ; Procedure ====================================================
2
3 ; End Procedure ================================================
```

Ну и вот тебе файлик с уже готовым проектом под этот шаблон [1]

AVR. Учебный курс. Операционная система. Введение.

Рано или поздно наступает момент когда сложность алгоритма становится такой, что дальнейшее развитие и усложнение программы превращается в нетривиальную задачу. Очень легко запутаться и тяжело отлаживать эту портянку. Многие бегут от этих сложностей в языки высокого уровня, впрочем это не особо спасает — разница минимальна на самом деле и проще не становится.
Самое верное решение в данном случае — внедрение в проект операционной системы. Которая бы предоставляла API для решения задач, а также обеспечивала порядок работы всей системы.

В результате было написано микроядро. Камрад **Serg2x2** подглядел концепцию в прошивке сотового телефона Motorola и портировал на микроконтроллер **AT89C2051**, после ее перенесли на AVR, а я привел все в библиотечный и структурированный вид, обвязал все удобными макросами, а также подробно описал и задокументировал. Так что теперь интеграция ядра операционки в проект под микроконтроллер **AVR** занимает буквально пару минут работы Copy-Paste.

Ядро обеспечивает очередь задач (пока без приоритетов, но это в планах) и службу таймеров. Многозадачность кооперативная, что накладывает соответствующие ограничения на стиль написания. Фактически, каждую процедуру мы пишем как прерывание, максимально коротко и быстро, все задержки вешая на службу таймеров.

Параметры и системные требования микроядра:

- Занимаемый обьем в Flash — **500 байт**, при желании можно ужать до **400 байт**, выкинув ненужные функции.
- Рекомендуемый объем RAM — не менее **20 байт**+стек, впрочем, можно еще ужать если нагрузка небольшая.
- Крайне желательная поддержка команд STS и LDS, можно и без них, но неудобно. Впрочем, макросы решают.

Суть всех заморочек
Представь что ты пишешь простейшую программу — мигнуть светодиодом. Как ты это будешь реализовывать? На ум сразу же приходит следующий алгоритм:

- Зажечь диод.
- Потупить в цикле
- Погасить диод.

Просто, логично, понятно.

А если усложнить? Мигать будем не одним диодом, а тремя, да еще чтобы первый мигал с частотой в 1кГц, второй в 500Гц, а третий 200Гц. Представил сразу же в голове алгоритм? Наверняка тут же получил переклин мозга и немудрено. Линейно развернуть такую структуру весьма сложно, даже если применить таймеры. Да, на таймерах выкрутишься, но это пока диодов три, а если десять?

Как то же самое делается на микроядре:

Главный цикл

- Вызов задачи 1
- Вызов задачи 3
- Вызов задачи 5
- Бесконечный цикл ожидания с опросом очереди

Таблица задач

- Задача_1: Зажечь диод 1. Поставить вызов задачи 2 в очередь с задержкой 1мс
- Задача_2: Погасить диод 1. Поставить вызов задачи 1 в очередь с задержкой 1мс
- Задача_3: Зажечь диод 2. Поставить вызов задачи 4 в очередь с задержкой 2мс
- Задача_4: Погасить диод 2. Поставить вызов задачи 3 в очередь с задержкой 2мс
- Задача_5: Зажечь диод 3. Поставить вызов задачи 6 в очередь с задержкой 5мс
- Задача_6: Погасить диод 3. Поставить вызов задачи 5 в очередь с задержкой 5мс

Все. Основной алгоритм получается практически линейным. Остальное делает ядро. Не меняя структуры можно добавить еще задач, навешать их в любом порядке. И это практически не повлияет на выполнение остального кода, разве что придется немного скорректировать настройки таймера.

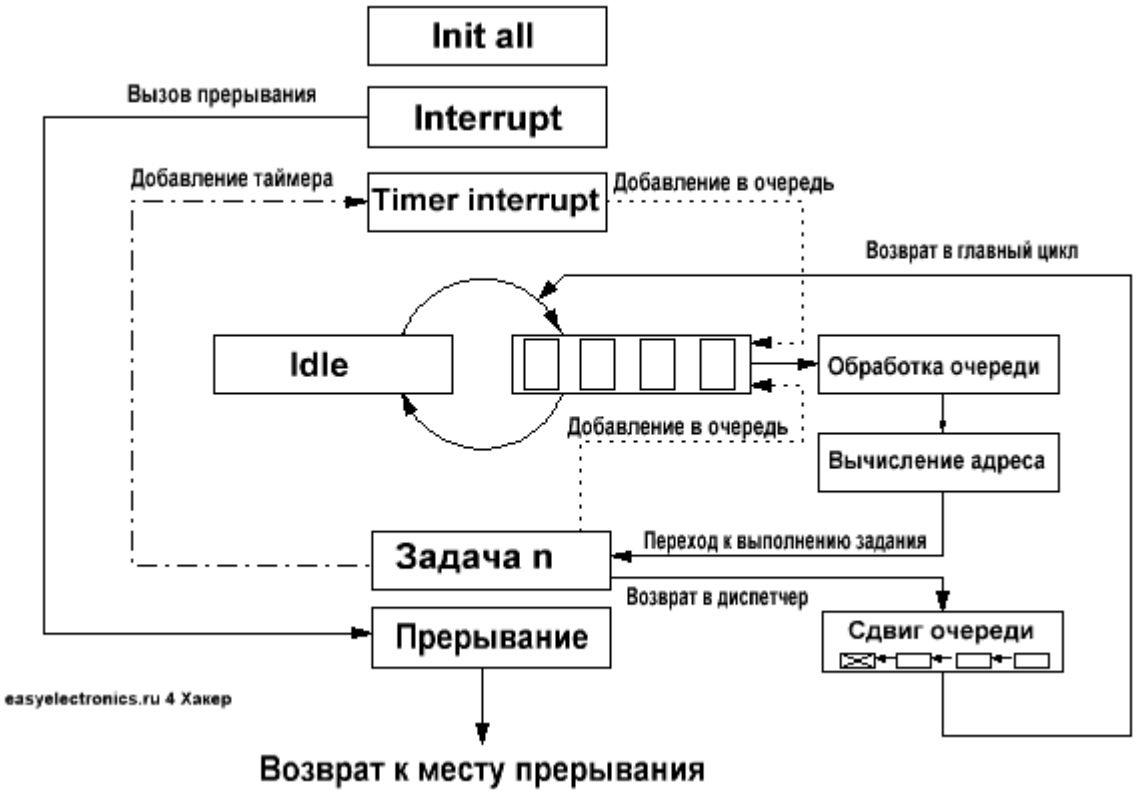
По договоренности с издательством Gameland я, до выхода журнала не буду раскрывать всех подробностей. В мартовском журнале Хакер будет статья по микроядру, после выхода журнала я еще немного подожду и со спокойной душой продолжу.

Пока же можете скачать готовую программу на микроядре [1] и посмотреть что там как. Может разберетесь :)

AVR. Учебный курс. Операционная система. Диспетчер задач.

В прошлой части [1] возник вопрос организации программы по задачам. Чтобы можно было разбить программу на кучу независимых частей и не заморачиваться на тот счет, что где то у нас будет затык.
Затык, конечно может быть, это все же не вытесняющая многозадачность с защищенным режимом, но разрулить все будет гораздо проще.

Общая диаграмма работы ОС



Что из себя представляет задача
Это практически то же самое, что и процедура, вызываемая командой RCALL с тремя отличиями:

- Вызывается она не мгновенно, а **в порядке очереди**.
- Вызов задачи идет не по ее адресу, а **по ее порядковому номеру** в таблице переходов.
- Возврат из нее идет не в то же место откуда вызывали, а **в цикл диспетчера задач**.

Сама задача представляет собой обычную процедуру, записанную без каких либо замудреностей.

Располагается там, где обычно прописаны вызываемые процедуры. В этом отношении ничего не поменялось. В принципе, вызывать можно код и из внешнего файла, главное правильно прописать все это в таблице переходов.

В тестовом примере это выглядит так:

Расположение: Trash-rtos.asm [2] — главный файл программы

```
1  ;=========================================================
2  ;Tasks
3  ;=========================================================
4  Idle:           RET     ; Задача пустого цикла, но ничего
5                          ; не мешает сунуть сюда любой код.
6                          ; Он будет выполнен. В последнюю очередь.
7  ;---------------------------------------------------------
8  Fire:           LDS     OSRG,U_B        ; Это код задачи "Fire"
9                  OUTI    UDR,'A'         ; Выдрано из реального проекта
10                                         ; Суть данного кода не важна
11                 CBI     PORTD,7         ; Поэтому можешь не вникать. Тут
12                 NOP                     ; Может быть абсолютно любой
13                 NOP                     ; код -- код твоей задачи!
14                 SBI     PORTD,7
15                                         ; Если любопытно, то тут обычное
16                 LDS     Counter,PovCT   ; заполнение сдвигового регистра
17                 LDPA    Lines           ; из трех микросхем 74HC164
18                                         ; средствами SPI передатчика
```

```
19                CLR     OSRG            ; Оставил его лишь для примера.
20                                        ; Чтобы наглядно показать, что
21                ADD     ZL,Counter      ; Из себя представляет задача.
22                ADC     ZH,OSRG
23
24                LPM     OSRG,Z+
25                OUT     SPDR,OSRG
26
27 Wait0:         SBIS     SPSR,SPIF
28                        RJMP    Wait0
29
30                INC     Counter
31
32                CPI     Counter,150
33                BRSH    Clear
34
35                STS     PovCT,Counter
36                RET
37
38 Clear:         CLR     Counter
39                STS     PovCT,Counter
40                RET                     ; Выход из задачи только по RET!!!
41 ;-----------------------------------------------------
42 Task2:         RET                     ; Это пустые заглушки. На этом месте
43                                        ; могла бы быть ваша задача! :)
44 ;-----------------------------------------------------
45 Task3:         RET                     ; Аналогично, надо будет задействую.
46 ;-----------------------------------------------------
47 Task4:         RET                     ; Названия в стиле Task4 тоже живут
48                                        ; недолго. Обычно переименовываю
49 ;-----------------------------------------------------
50 Task5:         RET                     ; Как с задачей "Fire"
51 ;-----------------------------------------------------
52 Task6:         RET
53 ;-----------------------------------------------------
54 Task7:         RET
55 ;-----------------------------------------------------
56 Task8:         RET
57 ;-----------------------------------------------------
58 Task9:         RET
```

Таблица переходов
После всего кода задач располагается таблица переходов и код самой ОС:

Расположение: Trash-rtos.asm [2] — главный файл программы, в самом низу, в конце ПЗУ

```
1  ;=====================================================================
2  ; RTOS Here
3  ;=====================================================================
4                 .include "kerneldef.asm"       ; Настройки ядра - переменные и ряд
5  макросов.
6                 .include "kernel.asm"          ; Подклчюаем ядро ОС.
7                               ;Это таблица переходов.
8  TaskProcs:     .dw Idle                ; [00]  Она содержит в себе реальные адреса задач
9                 .dw Fire                ; [01] Как видишь, 0 тут у задачи пустого цикла,
10                .dw Task2               ; [02] 01 у "Fire", ну и дальше
11                .dw Task3               ; [03] По порядку.
12                .dw Task4               ; [04]
13                .dw Task5               ; [05]
14                .dw Task6               ; [06]
15                .dw Task7               ; [07]
16                .dw Task8               ; [08]
```

```
                .dw Task9               ; [09]
```

Причем, в таблице задач не обязательно должны быть разные задачи. Можно делать одну, **но на разные ячейки**, например так:

```
1  TaskProcs:   .dw Idle                ; [00]
2               .dw Fire                ; [01]
3               .dw Task2               ; [02]
4               .dw Task3               ; [03]
5               .dw Task4               ; [04]
6               .dw Fire                ; [05]
7               .dw Task6               ; [06]
8               .dw Idle                ; [07]
9               .dw Idle                ; [08]
10              .dw Task9               ; [09]
```

Это иногда бывает удобно.
Таблица переходов нужна для того, чтобы можно было **любому адресу в программе дать адрес-смещение относительно начала таблицы перехода**. То есть, теперь, чтобы перейти на **Task4** нам не нужно знать точный адрес этой задачи, достаточно лишь знать, что ее адрес записан в таблице переходов в четвертой ячейке. Адрес начала таблицы переходов у нас фиксированный, поэтому просто прибавляем к нему смещение (равное номеру задачи*2) и берем оттуда искомый адрес. Благодаря этому, мы можем в очереди задач держать не двубайтные адреса переходов, а однобайтные номера под которыми эти адреса размещены в таблице.

А чтобы не путаться под каким номером какая задача спрятана, то введем для них символическое обозначение:
Расположение: kerneldef.asm [3] — файл макроопределений ядра

```
1  .equ TS_Idle    = 0     ; Просто нумерация. Не более того
2  .equ TS_Fire    = 1     ; Зато теперь можно смело отправлять в очередь
3  .equ TS_Task2   = 2     ; задачу с именем TS_Fire и не париться на счет того,
4  .equ TS_Task3   = 3     ; что запишется что то не то.
5  .equ TS_Task4   = 4     ; Тут все по порядку, жестко привязано к ячейкам таблицы!
6  .equ TS_Task5   = 5     ; Так что если в таблице и можно делать одинаковые задачи,
7  .equ TS_Task6   = 6     ; то тут у них идентификаторы должны быть разные!!!
8  .equ TS_Task7   = 7     ; А имена можно придумывать любые, они не привязаны ни к чему,
9  .equ TS_Task8   = 8     ; Главное самому не забыть что где.
10 .equ TS_Task9   = 9
```

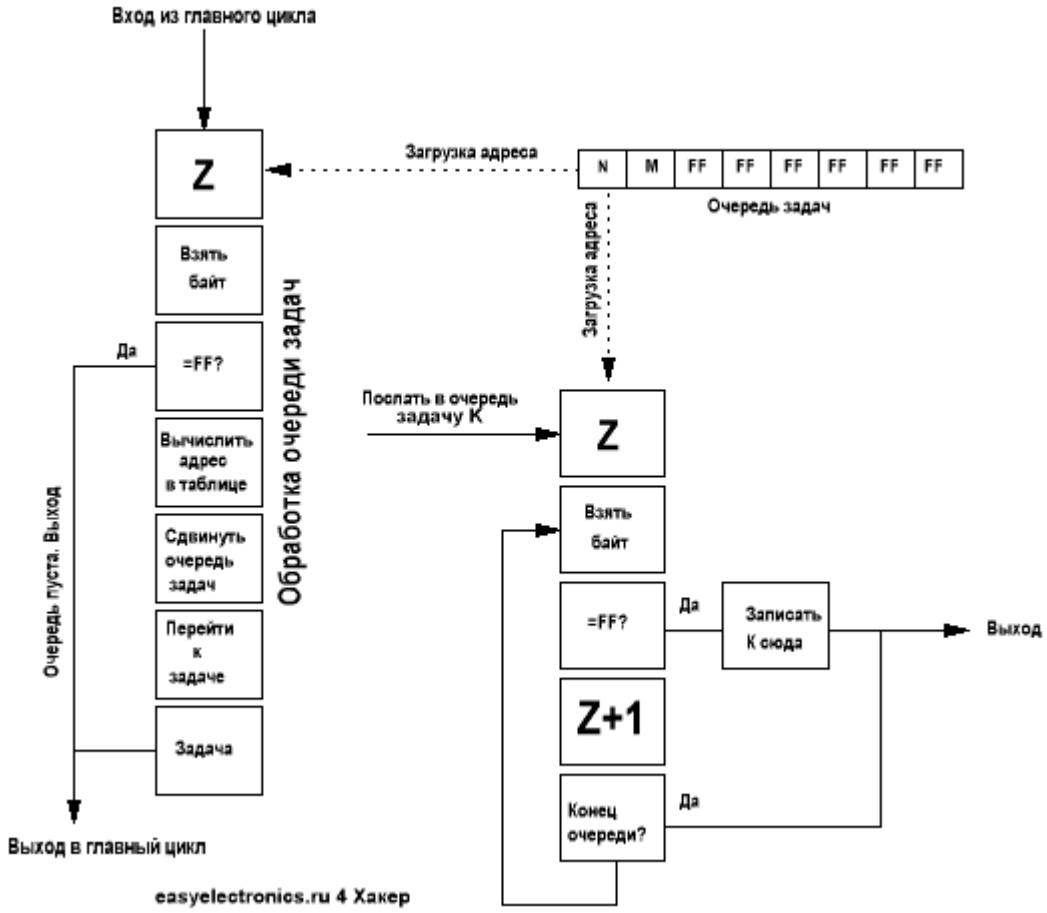
Очередь задач.
Логически выглядит как **строка в памяти**, где каждый байт это номер задачи. Два числа **0** и **FF** зарезервированы системой. 0 — это **Idle**, холостой цикл диспетчера. **FF** — нет задачи, конец очереди.

В коде это выглядит так:
Расположение: Trash-rtos.asm [2] — главный файл программы, самое начало. Где идет разметка ОЗУ

```
1               .DSEG
2               .equ TaskQueueSize      = 11            ; Длина очереди задач
3 TaskQueue:    .byte                   TaskQueueSize   ; Адрес очереди сотытий в SRAM
```

Длина очереди задается с запасом, чтобы не произошло ее срыва. Располагать ее лучше после остальных данных, чтобы она тянулась навстречу стеку.

Диспетчер задач
Это небольшая процедурка, которая берет из очереди байт, сравнивает его с FF. Если равно, значит очередь пуста и происходит выход в главный цикл. Если же там есть какое либо число отличное от FF, то оно загружается в регистр, происходит вычисление адреса по таблице переходов и прыжок на адрес задачи и выполнение кода. Перед прыжком текущий номер задачи удаляется из очереди, а вся очередь сдвигается на один байт вперед. При следующем заходе в диспетчер все повторяется заново до полного опустошения очереди. При этом в очередь можно добавлять новые задачи.



Главный цикл программы при этом выглядит следующим образом:
Расположение: Trash-rtos.asm [2] — главный файл программы

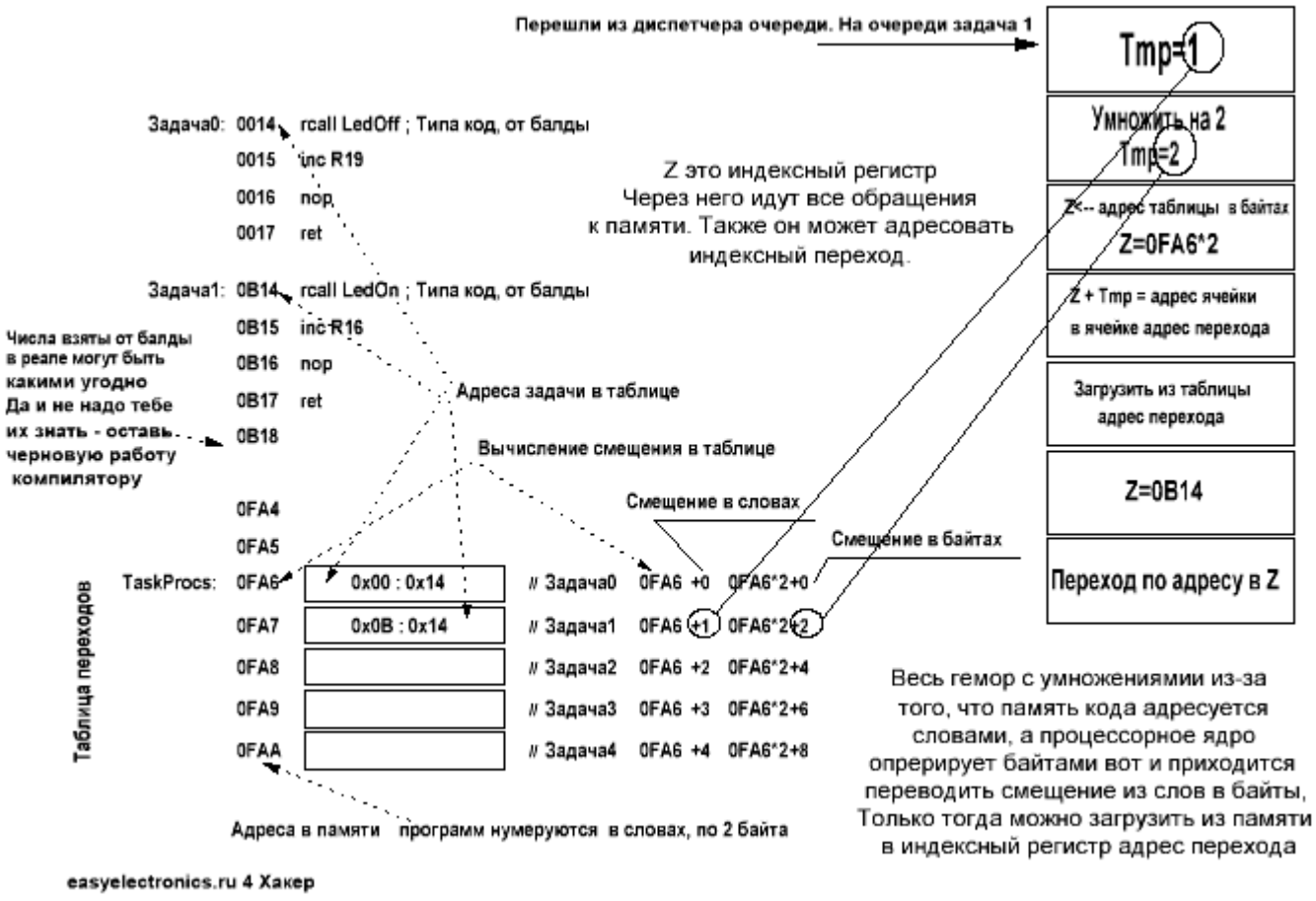
```
1 Main:           SEI                                 ; Разрешаем прерывания.
2                 WDR                                 ; Reset Watch DOG
3                 RCALL   ProcessTaskQueue            ; Обработка очереди процессов
4 (Диспетчер)
5                 RCALL   Idle                        ; Простой Ядра
                  RJMP    Main
```

Сам обработчик очереди несложен. Располагается в файле kernel.asm [4]

```
1  ProcessTaskQueue:
2                 ldi     ZL, low(TaskQueue)      ; Берем адрес начала очереди задач
3                 ldi     ZH, high(TaskQueue)     ; Напомню, что это в ОЗУ.
4
5                 ld      OSRG, Z         ; Берем первый байт (OSRG = R17 рабочий
6                 cpi     OSRG, $FF       ; регистр ОС) Сравниваем с FF
7                 breq     PTQL02         ; Равно? Значит очередь пуста - выход.
8
9                 clr     ZH              ; Сбрасываем старший байт
10                lsl     OSRG            ; А взятый номер задачи умножаем на 2
11                mov     ZL, OSRG        ; Так как адреса у нас двубайтные, а значит
12                                        ; И ячейки в таблице перехода двубайтные
13                                                ; Получается смещение по таблице
14
15                subi    ZL, low(-TaskProcs*2)  ; Прибавляем получившееся смещение к адресу
16                sbci    ZH, high(-TaskProcs*2)         ; начала таблицы переходов. Ну и
17 что, что AVR
18                                                ; не умеет
```

```
19                                              ; складывать регистр с числом + перенос.
20                                      ; Зато умеет вычитать, а минус на минус дают плюс!
21 :)
22                                      ; Математика царица всех наук!
23                                      ; Теперь в Z у нас адрес где лежит адрес перехода.
24
25                  lpm                 ; Берем этот адрес! Сначала в R0
26                  mov     OSRG, r0    ; Потом в OSRG
27                  ld      r0, Z+      ; Эта команда ничего ценного на грузит, мы ее
28 применили
29                                      ; Ради "Z+" чтобы увеличить адрес в Z и взять
30 второй байт
31                                      ; Целевого адреса по которому мы перейдем.
32                  lpm                 ; Берем в R0 второй байт адреса.
33
34                  mov     ZL, OSRG    ; А из OSRG перекладываем в ZL
35                  mov     ZH, r0      ; И из R0 в ZH. Теперь у нас в Z полный адрес
36 перехода.
37                                      ; Можно драпать, в смысле IJMP - индексный переход
38 по Z
39                                      ; Но пока рано! Надо же еще очередь в порядок
40 привести!
41                  push    ZL          ; Спрячем наш адрес, наше сокровище, в стек...
42                  push    ZH          ; Глубоко зароем нашу прелесссть....
43
44                                              ; Займемся грязной работой. Продвинем
45 очередь.
46                  ldi     Counter, TaskQueueSize-1; Загрузим длинну очереди. Иначе мы всю
47 память
48                  ldi     ZL, low(TaskQueue)      ; подвинем. И хапнем в Z начало очереди.
49                  ldi     ZH, high(TaskQueue)
50
51                  cli                 ; Запретим прерывания. А то если очередь сорвет
52 получим
53                                      ; армагедец.
54
55 PTQL01:          ldd     OSRG, Z+1   ; Грузим из следующего Z+1 байта и перекладываем
56                  st      Z+, OSRG    ; все в Z, а Z после этого увеличиваем на 1
57                  dec     Counter     ; Уменьшаем счетчик (там длинна очереди!)
58                  brne    PTQL01      ; Если не конец, то в цикле.
59                  ldi     OSRG, $FF   ; А если конец, то по последнему адресу записываем
60 FF
61                  st      Z+, OSRG    ; Который является признаком конца очереди.
62
                    sei                 ; Разрешаем прерывания. Можно расслабиться

                    pop     ZH          ; Достаем из стека нашу прелесссть... наш адрес
  перехода
                    pop     ZL          ; Оба байта, старший и младший.

                    ijmp    ; Переходим в задачу!!!
                            ; Обрати внимание - сюда мы пришли по RCALL из главного цикла
                            ; Значит в стеке у нас лежит адрес возврата. А ушли мы в задачу по
  IJMP
                            ; который стек не меняет. Но это не страшно! Ведь из задачи мы
                            ; выходим по RET!
  PTQL02:           ret
```

Для большей понятности нарисовал диаграммку со стрелочками разными, про то как формируется адрес в таблице переходов:



Задачи кладутся в очередь другой процедурой:

```
1               ldi     OSRG, TS_Task4          ; Запускаем в очередь задачу Task4
2               rcall   SendTask
```

Делать это можно где угодно, хоть в прерывании, хоть в другой задаче.
Для удобства был написан макрос:

```
1               SetTask  [task]
```

Сама процедура SendTask работает тоже несложно, она всего лишь ставит задачу в очередь.
Расположение: kernel.asm [4]

```
1  ; OSRG - Event                       ; В рабочем регистре ОС - номер задачи
2  SendTask:      push    ZL            ; Сохраняем все что используется
3                 push    ZH            ; в стеке
4                 push    Tmp2
5                 push    Counter
6
7                 in      Tmp2,SREG     ; Сохраняем значение флагов
8                 push    Tmp2
9
10                ldi     ZL, low(TaskQueue)     ; Грузим в Z адрес очереди задач.
11                ldi     ZH, high(TaskQueue)
12
13                ldi     Counter, TaskQueueSize ; А в счетчик длинну очереди, чтобы
14                                               ; не начать всю память засаживать.
15
16
```

```
17                cli                           ; запрещаем прерывания.
18
19 SEQL01:        ld      Tmp2, Z+              ; Грузим в темп байт из очереди
20
21                cpi     Tmp2, $FF       ; и ищем ближайшее пустое место = FF
22                breq    SEQL02          ; Если нашли, то переходим на сохранение
23
24                dec     Counter         ; Либо конец очереди по счетчику.
25                breq    SEQL03
26                rjmp    SEQL01
27
28 SEQL02:        st -Z,  OSRG            ; Нашли? Сохраняем в очереди номер задачи.
29
30 SEQL03: pop Tmp2               ; Возвращаем флаги. Если там прерывание было
31                out SREG,Tmp2           ; разрешено, то оно вернется в это значение.
32
33                pop Counter             ; Выходим, достав все заныченное.
34                pop Tmp2
35                pop ZH
36                pop ZL
37                ret
```

Архив с исходниками и работающим проектом для ATMega8 [5]

З.Ы.
Ухх, ну накатал телегу. Аж самому страшно. А там еще столько же, про таймерную службу, да пошаговые руководства по инсталляции и использованию… Но это после. Пока переварите это.

Продолжение следует…

AVR. Учебный курс. Операционная система. Таймерная служба

Третья часть ~~марлезонского балета~~ описалова самопальной операционной системы для AVR.

Итак, у нас есть очередь задач и общая логика работы системы. Но одной очереди задач с диспетчером мало. Нужно распределять задачи по времени, задавать интервалы, запускать отложенные задачи. Всем этим будет заниматься **служба таймеров**.

В чем ее суть ее работы:
Время разбивается на интервалы, скажем, по 1мс. Такой выдержки хватает для большинства задач. Также у нас должна быть очередь программных таймеров, размещенных в ОЗУ. На каждый таймер отводится **три байта**: Первый — идентификатор задачи. Два других — выдержка в миллисекундах.

Два байта позволяют организовать выдержку в 65.5 секунд. Конечно, можно сделать и больше, если отвести на временную выдержку три или даже четыре байта, но такие большие временные интервалы пригождаются редко, поэтому проще перехватиться через дополнительную переменную для конкретной задачи, а не нагружать таймерную службу обсчетом дополнительных байт.

Один из свободных аппаратных таймеров программируем на то, чтобы он **генерировал прерывание каждые 0.001с**

О работе аппаратных таймеров написано в разделе про программирование встроенной периферии. Пока можешь не заморачиваться. Суть лишь в том, что настроенный таймер тикает независимо от главной программы и в заданное время выдает прерывание.

По прерыванию мы берем из очереди таймеров первый байт и сравниваем его с 0xFF, за 0xFF принято неактивное состояние. Если же там не 0xFF, то значит это идентификатор задачи, а таймер активен. Поэтому берем третий байт, декрементируем его, если он стал равен нулю декрементируем второй байт и если оба байта не стали равны

нулю переходим к проверке следующего байта. В случае если время истекло, то идентификатор задачи пихается в очередь задач на исполнение.

Обработчик прерывания таймера:

```
                push    OSRG            ; Прячем OSRG в стек
                in      OSRG,SREG       ;
                push    OSRG            ; Сохранение регистра OSRG и регистра состояния
1
 SREG
2
3
                push    ZL
4
                push    ZH                      ; сохранение Регистра Z
5
                push    Counter                 ; сохранение Регистра Counter
6
7
                ldi     ZL,low(TimersPool)      ; Загрузка с регистр Z адреса таймерной
8
 очереди,
9
                ldi     ZH,high(TimersPool)     ; по которому находится информация о
10
 таймерах
11
12
                ldi     Counter,TimersPoolSize          ;  Берем максимальное количество
13
 таймеров
14
15
 Comp1L01:      ld      OSRG,Z                  ; OSRG = [Z] ; Получить номер события
16
                cpi     OSRG,$FF                ; Проверить на "NOP = FF"
17
                breq    Comp1L03                ; Если NOP то переход к следующей позиции
18
19
                clt                     ; Флаг T используется для информации об окончании
20
 счёта
21
                ldd     OSRG,Z+1        ; Грузим в OSRG первый байт времени
22
                subi    OSRG,low(1)     ; Уменьшение младшей части счётчика на 1
23
                std     Z+1,OSRG        ; И сохраняем ее обратно туда откуда взяли
24
                breq    Comp1L02        ; Если образовался 0 то флаг T не устанавливаем
25
                set                     ; А если байт не закончился, то ставим T
26
27
 Comp1L02:      ldd     OSRG,Z+2        ; Берем второй байт времени.
28
                sbci    OSRG,High(1)    ; Уменьшение старшей части счётчика на 1
29
                std     Z+2,OSRG        ; Сохраняем где взяли
30
                brne    Comp1L03        ; Счёт не окончен
31
                brts    Comp1L03        ; Счёт не окончен (по T)
32
33
                ld      OSRG,Z          ; Получить номер задачи
34
                rcall   SendTask                ; послать в системную очередь задач
35
36
                ldi     OSRG,$FF        ; = NOP (задача выполнена, таймер самоудаляется)
37
                st      Z, OSRG         ; Прописываем в заголовок таймера FF
38
39
 Comp1L03:      subi    ZL,Low(-3)      ; Пропуск таймера.
40
                sbci    ZH,High(-3)     ; Z+=3 - переход к следующему таймеру
41
                dec     Counter         ; счетчик таймеров
42
                brne    Comp1L01        ; Если это был не последний таймер, то еще раз
43
44
                pop     Counter         ; восстанавливаем переменные
45
                pop     ZH
46
                pop     ZL
47
48
                pop     OSRG            ; Восстанавливаем регистры
49
                out     SREG,OSRG
                pop     OSRG
                RETI                    ; Выход из прерывания таймера
```

Постановка таймера.
При постановке таймера мы вначале шерстим всю очередь таймеров на предмет наличия там записи с таким же

идентификатором. Если находим, то обновляем его новым значением. Если не находим, то записываем следующе значение.

Постановка делается макросом из файла kernel_macro.asm

```
1               SetTimerTask    [task],[time]
```

Сам макрос развертывается в такой код:

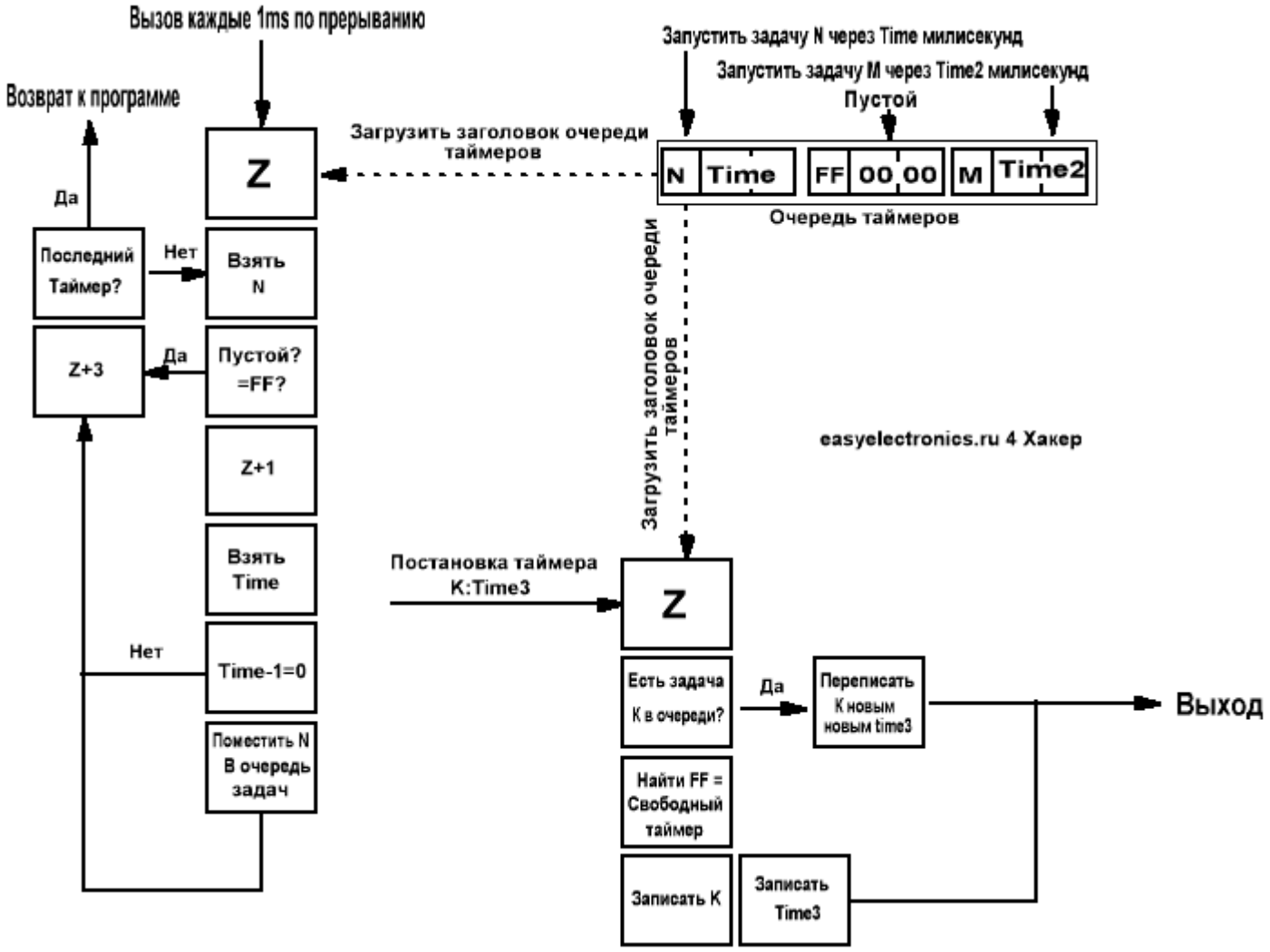
```
1         ldi     OSRG, [Task]
2         ldi     XL, Low([Time])               ; Задержка в милисекундах
3         ldi     XH, High([Time])                      ; От 1 до 65535
4         rcall   SetTimer
```

Как видим, тут используется регистровая пара X и вызывается функция постановки таймера. Про использование ресурсов в этих макросах и процедурах надо помнить и сохранять их в стеке если постановка таймера идет из прерывания.

Сама функция **SetTimer** работает просто:
Расположение: kernel.asm

```
1   SetTimer:                       ; В OSRG номер задачи. В X время
2           push    ZL              ; Сохраняем все что используем
3           push    ZH
4           push    Tmp2
5           push    Counter
6
7           ldi     ZL, low(TimersPool)     ; Берем адрес очереди таймеров
8           ldi     ZH, high(TimersPool)
9
10          ldi     Counter, TimersPoolSize         ; Берем число таймеров
11
12  STL01:  ld      Tmp2, Z         ; Хватаем первый заголовок
13          cp      Tmp2, OSRG      ; Сравниваем с тем который хотим записать
14          breq    STL02           ; Если такой уже есть, идем на апдейт
15
16          subi    ZL, Low(-3)     ; Выбираем следующий
17          sbci    ZH, High(-3)    ; Z+=2
18
19          dec     Counter         ; Уменьшаем счетчик
20          breq    STL03           ; Если ноль переход к записи нового таймера
21          rjmp    STL01
22
23  STL02:                          ; Если нашли такой же, то делаем ему апдейт
24          std     Z+1, XL         ; Значения временем из X
25          std     Z+2, XH         ; Оба байта
26          rjmp    STL06           ; Выходим из процедуры
27
28  STL03:                                  ; Если аналогичного не нашли
29          ldi     ZL, low(TimersPool)     ; То делаем добавление нового
30          ldi     ZH, high(TimersPool)    ; Заново берем адрес очереди
31
32          ldi     Counter, TimersPoolSize         ; И ее длинну
33
34  STL04:  ld      Tmp2, Z         ; Хватаем первый заголовок
35          cpi     Tmp2, $FF       ; Пуст?
36          breq    STL05           ; Переходим к записи таймера
37
38          subi    ZL, Low(-3)     ; Если не пуст выбираем следующий таймер
39          sbci    ZH, High(-3)    ; Z+=2
40
```

```
41          dec     Counter         ; Очередь кончилась?
42          breq    STL06           ; Да. Нет таймеров свободных. Увы. Выход
43                                  ; Краша не будет, но задача не выполнится
44          rjmp    STL04           ; Если очередь не вся, то повторяем итерацию
45
46  STL05:  cli                     ; Запрет прерываний перед записью в очередь
47          st      Z, OSRG         ; Сохраняем новый таймер
48          std     Z+1, XL         ; И его время
49          std     Z+2, XH
50          sei                     ; Разрешаем прерывания
51
52  STL06:                          ; Выходим, достав все из стека.
53          pop     Counter
54          pop     Tmp2
55          pop     ZH
56          pop     ZL
57          ret
```



Вот, ничего сложного. Из кода сразу же понятны недостатки данного алгоритма.
Время выполнения **зависит от числа таймеров и плавает**, особенно на малых выдержках в 1-2мс. Так что точные замеры времени ей поручать нельзя. Для этого придется задействовать другой аппаратный таймер и все нежные манипуляции делать на нем. Но на выдержке в 500мс, глядя осциллографом на тестовый импульс, я особых искажений в показаниях не заметил. Т.к. AVR щелкает команды очень быстро и чем быстрей тактовая частота, тем меньше влияние числа таймеров на временную выдержку (растет отношение холостых тактов таймера к времени выполнения процедуры таймерной очереди).
Малые временные интервалы, меньшие чем 1мс этому таймеру тоже недоступны. Конечно, можно взять и понизить планку, сделать срабатывание прерывания не каждую миллисекунду, а каждые 500мкс. Но тут падает точность. Так что если такое потребуется, то делать это на другом таймере.

Но, в целом, несмотря на недостатки, очень удобная служба получилась. А главное вытыкается в момент. Прикол с апдейтом таймера кажется лишним, но реально часто пригождается. Например, когда по условию надо отложить событие. Берешь и перезаписываешь таймер, подобно программному вачдогу. А если надо две одинаковые задачи по таймеру сделать в разное время, то никто не запрещает добавить ее в таблицу переходов на новый идентификатор и будет тебе профит.

Продолжение следует...

AVR. Учебный курс. Операционная система. Установка

Ядро у нас есть, теперь осталось это все хозяйство запихать на МК. Для этого всего лишь надо рассовать нужные части кода в исходник. Показывать буду на примере ATmega8. Для других МК разница минимальная. Может быть с таймером что нибудь помудрить придется, но не более того.
Например, недавно, вкорячивал ту же схему на ATmega168, так пришлось подправить инцилизацию таймера — регистры там зовутся по другому. Пришлось изменить макрос OUTI — так как многие привычные уже регистры перестали загружаться через комадну OUT — выпали из диапазона, только через LDS/STS ну и, собственно, все хлопоты. Потратил минут 20 на переименование регистров и заработало.

Итак. Есть у нас совершенно новый пустой файл NewMega8-rtos.asm

```
1  ; Добавляем в него первым же делом инклюдник восьмой меги:
2
3                 .include "m8def.inc"   ; Используем ATMega8
4
5  ; Следом я добавляю файл с моими макроопределениями в котором записаны все символические
6  имена
7  ; для ресурсов, вроде регистров, портов, отдельных пинов. ИМХО удобней все держать по
8  разным
9  ; файлам, но тут уже дело за вашими привычками.
10
11                .include "define.asm"
12
13 ;Потом файл с макросами, он тоже отдельный и кочует из программы в программу. Именно там
14 всякие
15 ; левые самодельные команды вроде OUTI прописаны.
16
17                .include "macro.asm"   ; Все макросы у нас тут
18
19 ; Следом идет файл макросов ядра ОС. Он должен располагаться в начале программы, иначе
20 компилятор
21 ; не поймет. Именно там прописаны макросы таймерной службы, добавления задачи и таймера,
22 там же
23 ; заныкан стандартный макрос инциализации UART и многое другое.
24
25                .include "kernel_macro.asm"
26
27 ;Дальше прописывается сегмент оперативной памяти в котором заранее определены
28 ; все очереди задач и таймеров.
29                .DSEG
30                .equ    TaskQueueSize = 11     ; Размер очереди событий
31 TaskQueue:     .byte   TaskQueueSize          ; Адрес очереди сотытий в SRAM
32                .equ    TimersPoolSize = 5     ; Количество таймеров
33 TimersPool:    .byte   TimersPoolSize*3       ; Адреса информации о таймерах
34
35 ; Следом уже идет код, начинается кодовый сегмент. Надо заметить, что у меня вся таблица
36 ; Прерываний спрятана в vectors.asm и вместо таблицы в коде .include "vectors.asm" это
37 удобно
38
39                .CSEG
40                .ORG    0x0000         ; Проц стартует с нуля, но дальше идут вектора
```

```
41                RJMP    Reset
42
43                .ORG    INT0addr        ; External Interrupt Request 0
44                RETI
45                .ORG    INT1addr        ; External Interrupt Request 1
46                RETI
47
48        .ORG    OC2addr                 ; Timer/Counter2 Compare Match
49        RJMP    OutComp2Int             ;<<<<<<<< Прерывание ОС!!!
50
51                .ORG    OVF2addr        ; Timer/Counter2 Overflow
52                RETI
53                .ORG    ICP1addr        ; Timer/Counter1 Capture Event
54                RETI
55                .ORG    OC1Aaddr        ; Timer/Counter1 Compare Match A
56                RETI
57                .ORG    OC1Baddr        ; Timer/Counter1 Compare Match B
58                RETI
59                .ORG    OVF1addr        ; Timer/Counter1 Overflow
60                RETI
61                .ORG    OVF0addr        ; Timer/Counter0 Overflow
62                RETI
63                .ORG    SPIaddr         ; Serial Transfer Complete
64                RETI
65                .ORG    URXCaddr        ; USART, Rx Complete
66                RJMP    Uart_RCV
67                .ORG    UDREaddr        ; USART Data Register Empty
68                RETI
69                .ORG    UTXCaddr        ; USART, Tx Complete
70                RJMP    Uart_TMT
71                .ORG    ADCCaddr        ; ADC Conversion Complete
72                RETI
73                .ORG    ERDYaddr        ; EEPROM Ready
74                RETI
75                .ORG    ACIaddr         ; Analog Comparator
                  RETI
                  .ORG    TWIaddr         ; 2-wire Serial Interface
                  RETI
                  .ORG    SPMRaddr        ; Store Program Memory Ready
                  RETI
                  .ORG    INT_VECTORS_SIZE       ; Конец таблицы прерываний
```

После таблицы векторов идут обработчики прерываний. Они короткие, поэтому их размещаю в начале. Первым же обработчиком идет обработчик прерывания от таймера на котором висит таймерная служба ОС. Под таймерную службу желательно отдать самый стремный таймер, на который не завязана ШИМ или еще какая полезная служба. В идеале бы под это дело пустить Timer0, как самый лоховский. Но он не умеет считать от 0 до регистра сравнения, только от нуля до 255, впрочем, в обработчик прерывания можно добавить предварительную загрузку таймера0 нужным значением и не расходовать более навороченный таймер. Мне было лень, я повесил все на Таймер2, а в регистр сравнения прописал такое значение, чтобы прерывание было ровно один раз в 1мс. Разумеется, выставив предварительно нужный делитель.

```
1  ; Interrupts procs
2  ; Output Compare 2 interrupt  - прерывание по совпадению TCNT2 и OCR2
3  ; Main Timer Service - Служба Таймеров Ядра - Обработчик прерывания
4
5  OutComp2Int:   TimerService   ; Служба таймера OS
6                                ; Весь код обработчика в виде одного макроса
7                                ; Просто вставил и все. Куда угодно. Можно извратиться
8                                ; Подать импульсы с нужной частотой на какой-нибудь
9                                ; INT0 и службу таймеров повесить на его прерывание
10                               ; Разумеется, в таблице векторов из вектора  прописан
11                               ;переход сюда
12                RETI           ; выходим из прерывания
```

```
13
14 Uart_RCV:       RETI            ; Другие прерывания если нужны
15 Uart_TMT:       RETI
```

То все было обязательной подготовкой и разметкой адресов и памяти, а вот тут уже начинается сама программа. Именно отсюда стартует проц. Вписываем следующий код:

```
1 Reset:          OUTI    SPL,low(RAMEND)                ; Первым делом инициализируем стек
2                 OUTI    SPH,High(RAMEND)
```

Все инициализации у меня спрятана в **.include «init.asm»**, но тут я распишу ее полностью.

```
1  ; init.asm
2  ; Очистка памяти
3  RAM_Flush:     LDI     ZL,Low(SRAM_START)
4                 LDI     ZH,High(SRAM_START)
5                 CLR     R16
6  Flush:         ST      Z+,R16
7                 CPI     ZH,High(RAMEND)
8                 BRNE    Flush
9
10                CPI     ZL,Low(RAMEND)
11                BRNE    Flush
12
13                CLR     ZL
14                CLR     ZH
15
16 ; Init RTOS                    ; В исходнике все сделано
17 ;              INIT_RTOS       ; вот так вот, одним макросом. Макрос описан
18 ;                              ; в файле kernel_macro.asm
19 ;                              ; Но я распишу тут подробно. Там идет настройка
20 ;                              ; Таймера в работу
21
22 ; Содержимое макроса INIT_RTOS
23
24                OUTI    SREG, 0         ; Сброс всех флагов
25
26                rcall ClearTimers       ; Очистить список таймеров РТОС
27                rcall ClearTaskQueue    ; Очистить очередь событий РТОС
28                sei                     ; Разрешить обработку прерываний
29
30 ; Настройка таймера 2 - Основной таймер для ядра
31
32                .equ    MainClock       = 8000000               ; CPU Clock
33                .equ    TimerDivider    = MainClock/64/1000     ; 1 mS
34
35                OUTI    TCCR2,1<<CTC2|4<<CS20  ; Установить режим CTC и предделитель =64
36                OUTI    TCNT2,0                ; Установить начальное значение счётчиков
37
38
39                ldi OSRG,low(TimerDivider)
40                out OCR2,OSRG                  ; Установить значение в регистр сравнения
41 ; Конец макроса INIT_RTOS
42
43
44                OUTI    TIMSK,1<<OCF2          ; Разрешить прерывание по сравнению
45
46 ; Инициализация остальной периферии
47                USART_INIT
48 ; Конец init.asm
```

После инициализации идет секция запуска фоновых приложений. Добавим пока меточку про запас, с нас не убудет.

```
1 Background:    NOP            ; Пока тут ничего нет
```

Главный цикл. Его надо скопировать без изменений, как есть.

```
1 Main:   SEI                          ; Разрешаем прерывания.
2         wdr                          ; Reset Watch DOG
3         rcall   ProcessTaskQueue     ; Обработка очереди процессов
4         rcall   Idle                 ; Простой Ядра
5         rjmp    Main                 ; Основной цикл микроядра РТОС
6
7 ; В Idle можно сунуть что нибудь простое, быстрое и некритичное.
8 ; Но я обычно оставляю его пустым.
```

После главного цикла вставляется шаблон под секцию задач. Именно сюда вписывается наш исполняемый код. Тут творится самое интересное

```
1  Idle:          RET
2  ;---------------------------------------------------------------------------
3  Task1:         RET
4  ;---------------------------------------------------------------------------
5  Task2:         RET
6  ;---------------------------------------------------------------------------
7  Task3:         RET
8  ;---------------------------------------------------------------------------
9  Task4:         RET
10 ;---------------------------------------------------------------------------
11 Task5:         RET
12 ;---------------------------------------------------------------------------
13 Task6:         RET
14 ;---------------------------------------------------------------------------
15 Task7:         RET
16 ;---------------------------------------------------------------------------
17 Task8:         RET
18 ;---------------------------------------------------------------------------
19 Task9:         RET
20
21 ; А после секции задач вставляем шаблонную таблицу переходов и код ядра
22                .include "kerneldef.asm"      ; Подключаем настройки ядра
23                .include "kernel.asm"         ; Подклчюаем ядро ОС
24
25 TaskProcs:     .dw Idle              ; [00]
26                .dw Task1             ; [01]
27                .dw Task2             ; [02]
28                .dw Task3             ; [03]
29                .dw Task4             ; [04]
30                .dw Task5             ; [05]
31                .dw Task6             ; [06]
32                .dw Task7             ; [07]
33                .dw Task8             ; [08]
34                .dw Task9             ; [09]
```

Готово! Можно компилировать, пока это пустой проект. Но ненадолго.

Итак, теперь то же самое, но по пунктам.
Установка AVR OS

- Создаем пустой проект
- Вставляем файлы макроопределений

- Вставляем разметку памяти под очереди задач/таймеров
- Вставляем таблицу векторов прерываний
- Прописываем в таблице векторов прерываний переход на обработчик таймера по переполнению
- Добавляем обработчик прерываний
- Прописываем стартовую метку и инициализацию стека
- Инициализация всего что только можно — портов, периферии, обнуление ОЗУ, обнуление очередей, запуск таймера ОС.
- Добавляем секцию фоновых задач
- Добавляем код главного цикла
- Добавляем шаблонную сетку задач
- Добавляем код ядра и таблицу переходов.
- Пишем наш код
- . . .
- PROFIT

В следующий раз я покажу практический пример работы с этой ОС. В котором будет красочно показано ради чего, собственно, этот геморрой и почему мне он так нравится :)

AVR. Учебный курс. Операционная система. Пример.

Отлично, с теорией работы ОС ознакомил. Устанавливать научил, осталось научить использовать весь этот конвеерно таймерный шухер. Чем я сейчас и займусь. Сразу берем быка за рога и формулируем учебно-боевую программу.
Тестовое задание:
Пусть у нас будет **ATMega8**, с несколькими кнопками. АЦП и подключеним к компу через UART. На меге будет три светодиода.

- Девайс должен при включении начинать мигать зеленым диодом, мол работаю.
- При этом раз в секунду сканировать показания АЦП и если показания ниже порога — Моргать красным диодом.
- По сигналу с UART с целью защиты от ошибок сделать по байту ‘R’ установку флага готовности, а потом, в течении 10ms если не придет байт ‘A’ сбросить флаг готовности и игнорировать все входящие байты кроме ‘R’. Если ‘A’ придет в течении 10мс после ‘R’, то отправить в UART ответ и зажечь белый диод на 1 секунду.

Вот так вот, не сильно сложно. Но мне просто лень делать что либо сложней, а для тестовой задачи сгодится.
Итак, что у нас есть:
Две фоновые задачи, которые выполняются всегда и постоянно:

- Мигать зеленым диодом
- Проверять показания с АЦП

И цепочки:

- Прерывание АЦП — Проверка условия — зажечь диод — погасить диод.
- Прерывание UART- Проверить на R — взвести флаг (ждать ‘A’ — зажеть диод — погасить диод) — снять флаг

Описываем задачи:

- OnGreen — зажечь зеленый
- OffGreen — погасить зеленый
- OnRed — зажечь красный
- OffRed — погасить красный
- OnWhite — зажечь бело-лунный
- OffWhite — погасить бело-лунный
- Reset_R — сбросить флаг готовности
- ADC_CHK — проверить АЦП.

Собственно я это так, для ясности. Обычно я по ходу дела обвешиваю нужным функционалом. Инициализацию портов и всего прочего я опущу — не маленькие уже, DDR всякие сами выставите. Буду указывать код кусками, с пояснениями что где, а всю картину увидите в исходнике.

Итак, начинаем наращивать мясо:
Добавляем первую задачу — **Мигать зеленым диодом.**Для этого любую удобную секцию из раздела Task берем и переименовываем по своему вкусу. Будем по порядку. Под раздачу пойдет первая — Task1. Но сначала надо прописать ее номер в дефайнах. Поэтому сразу же лезем в **kerneldef.asm** и вписываем там:

```
1 .equ TS_Idle    = 0     ;
2 .equ TS_OnGreen= 1     ;  <<<<
3 .equ TS_Task2   = 2     ;
4 .equ TS_Task3   = 3     ;
5 . . .
```

Затем возвращаемся к таблице задач. И в поле **Task1** вписываем нашу задачу. Переименовывая метку (это не обязательно, но иначе запутаетесь в этих бесконечных **Task-n**.
Сразу же описываем саму задачу — она простая, просто зажечь диод. Одна команда процессора — SBI
Следом идет макрос **SetTimerTask**, считай это командой **API** нашей ОС. Значит что спустя 500мс надо выполнить задачу **OffGreen**

```
1 ; Tasks
2 Idle:           RET
3 ;-------------------------------------------------------------------------------
4 OnGreen:        SBI     PORTB,1         ; Зажечь зеленый
5                 SetTimerTask    TS_OffGreen,500
6                 RET
7 ;-------------------------------------------------------------------------------
8 Task2:          RET
9                 . . .
```

Но у нас нет такой задачи! Не вопрос, добавляем!
Прописываем сначала в дефайнах:

```
1 .equ TS_Idle           = 0     ;
2 .equ TS_OnGreen        = 1     ;
3 .equ TS_OffGreen       = 2     ; <<<<
4 . . .
```

Потом добавляем ее код в область задач

```
1  ; Tasks
2  Idle:           RET
3  ;-------------------------------------------------------------------------------
4  OnGreen:        SBI     PORTB,1         ; Зажечь зеленый
5                  SetTimerTask    TS_OffGreen,500
6                  RET
7  ;-------------------------------------------------------------------------------
8  OffGreen:       CBI     PORTB,1         ; Погасить зеленый
9                  SetTimerTask    TS_OnGreen,500
10                 RET
11 ;-------------------------------------------------------------------------------
12 Task3:          RET
13                 . . .
```

Видишь **первая задача ссылается на вторую, а вторая на первую**. Одна зажигает, вторая гасит. В итоге, они будут кольцом запускать друг друга, а зеленый диод, повешанный на PB1 будет мигать с интервалом 0.5с.

Теперь надо вписать их в таблицу переходов каждого на свой номер в **kerneldef.asm**:

```
1 TaskProcs:      .dw     Idle            ; [00]
2                 .dw     OnGreen         ; [01] TS_OnGreen
3                 .dw     OffGreen        ; [02] TS_OffGreen
4                 .dw     Task3           ; [03]
5                 . . .
```

Отлично. Фоновая задача сформирована, теперь надо только ее стартануть. Помните секцию **Background**? Вот я ее держу именно для этих случаев. Так то можно откуда угодно сделать наброс. Но удобней это делать из одного места.
Вот там и делаем:

```
1 Background:     RCALL   OnGreen
```

Задача все равно сформирована как процедура, так что как зайдет так и выйдет, а таймер запустится и дальше по цепи. То есть задачу можно стартануть простым RCALL. Тогда она первый раз выполнится мгновенно. А можно и через отдельный макрос **SetTask** будет выглядеть так:

```
1 Background:     SetTask TS_OnGreen
```

И выполнится в порядке общей очереди. Т.к. проц только стартовал и очеред пуста, то практически сразу же.
или по таймеру.

```
1 Background:     SetTimerTask TS_OnGreen,10000
```

Тогда мигать начнет спустя 10 секунд после запуска. Обрати внимание на то, что прямым **RCALL** мы указываем на метку, а через API мы передаем идентификатор задачи.

Да, не помешает напомнить, что из себя представляют макросы **SetTimerTask** и **SetTask**

```
1                 .MACRO SetTask
2                 ldi OSRG, @0            ; В OSRG номер задачи
3                 rcall SendTask          ; через событийный диспетчер
4                 .ENDM
5
6
7                 .MACRO SetTimerTask
8                 ldi     OSRG, @0        ; В OSRG номер задачи
9                 ldi     XL, Low(@1)     ;
10                ldi     XH, High(@1)    ; Задержка в милисекундах
11                rcall   SetTimer        ; поставить таймер в очередь
12                .ENDM
```

Видишь тут используется **OSRG** (R17) и пара **X**. Для задания времени. А также функция **SetTimer**. Функция безопасна — она все значения сохраняет в стеке, а вот макрос нет — при вызове макроса **убивается X и OSRG**. Обычно это не критично, но это надо знать и помнить. Особенно когда вызваешь это дело из прерываний.
А еще я показал тебе подробно эти макросы с целью намекнуть на то, что задачи можно ставить не только вручную, а еще и программным способом.
Например, брать из памяти цепочки идентификаторов задач и передавать их через OSRG в SendTask. Получится цифровая мегашарманка :) Главное «вращать барабан» не быстрей чем они выполняются, а то очередь сорвет. А там и до виртуальной машины не далеко… Хотя нет, к черту. Java программистов надо убивать апстену! =)

А дальше в том же ключе:
Добавляем задачу проверки АЦП. Разумеется прописываем ей TS номер в дефайнах, не буду показывать.

```
1  ; Tasks
2  Idle:          RET
3  ;-----------------------------------------------------------------------------------
4  OnGreen:       SBI     PORTB,1         ; Зажечь зеленый
5                 SetTimerTask   TS_OffGreen,500
```

```
6                RET
7  ;-----------------------------------------------------------------------------
8  OffGreen:     CBI     PORTB,1         ; Погасить зеленый
9                SetTimerTask   TS_OnGreen,500
10               RET
11 ;-----------------------------------------------------------------------------
12  ADC_CHK:     SetTimerTask   TS_ADC_CHK,1000
13               OUTI    ADCSRA,1<<ADEN|1<<ADIE|1<<ADSC|3<<ADPS0
14               RET
```

Как видишь, она сама себя запускает каждые 1000мс, т.е. каждую секунду. А стартует там же, из секции **Background**

```
1 Background:    RCALL   OnGreen
2                RCALL   ADC_CHK
```

Осталось дождаться прерывания, поэтому ставим в код прерывание по выполнению АЦП:

Кладу его рядом с остальными прерываниями:

```
1  ADC_OK:       push    OSRG
2                in      OSRG,SREG       ; Спасаем OSRG и флаги.
3                push    OSRG
4
5                IN      OSRG,ADCH       ; Взять показание АЦП
6                CPI     OSRG,Treshold   ; Сравнить с порогом
7                BRSH    EXIT_ADC        ; Если не достигнут выход
8
9                SetTask TS_RedOn        ; Запускаем мырг красным
10
11 EXIT_ADC:     pop     OSRG            ; Восстанавливаем регистры
12               out     SREG,OSRG
13               pop     OSRG
14               RETI                    ; Выходим из прерывания
```

Появилась еще одна задача — зажечь красный. Не вопрос, добавляем, прописав везде где нужно. Я же тут укажу только исполнительную часть. Сразу же впишу и задачу гашения красного. Чтобы уж в одном флаконе.

```
1  ; Tasks
2  Idle:         RET
3  ;-----------------------------------------------------------------------------
4  OnGreen:      SBI     PORTB,1         ; Зажечь зеленый
5                SetTimerTask   TS_OffGreen,500
6                RET
7  ;-----------------------------------------------------------------------------
8  OffGreen:     CBI     PORTB,1         ; Погасить зеленый
9                SetTimerTask   TS_OnGreen,500
10               RET
11 ;-----------------------------------------------------------------------------
12  ADC_CHK:     SetTimerTask   TS_ADC_CHK,1000
13               OUTI    ADCSRA,1<<ADEN|1<<ADIE|1<<ADSC|3<<ADPS0
14               RET
15 ;-----------------------------------------------------------------------------
16 OnRed:        SBI     PORTB,2
17               SetTimerTask   TS_OffRed,300
18               RET
19 ;-----------------------------------------------------------------------------
20 OffRed:       CBI     PORTB,2
21               RET
```

Как видишь, Красный зажигается от пинка с прерывания АЦП, а гаснет по собственному пинку через 300мс. Так как АЦП проверяется раз в секунду, то если порог будет ниже, то каждую секунду будет вызов задачи RedOn и светодиод будет моргать 0.3 секундной вспышкой. Если сделать длительность вспышки больше чем частота вызова прерывания АЦП, то диод будет просто гореть, так как прерывание АЦП будет постоянно обновлять ему таймер. До тех пор пока входное напряжение на АЦП не будет выше порога, тогда диод моргнет на свою выдержку и затихнет.

Так, что там у нас следующее? **UART**? Ну не вопрос! Считаем, что **UART** у нас уже проинициализирован и готов к работе. Не верите? Глянте в секцию **init.asm** Так что добавляем прерывание на прием:

```
1  Uart_RCV:       push    OSRG
2                  in      OSRG,SREG       ; Спасаем OSRG и флаги.
3                  push    OSRG
4                  PUSH    XL              ; SetTimerTask юзает X!!!
5                  PUSH    XH              ; Поэтому прячем его!
6                  PUSH    Tmp2            ; Ну и Tmp2 нам пригодится
7
8                  IN      OSRG,UDR
9                  CPI     OSRG,'R'                ; Проверяем принятый байт
10                 BREQ    Armed           ; Если = R  - идем взводить флаг
11
12                 LDS     Tmp2,R_flag     ; Если не R и флага готовности нет
13                 CPI     Tmp2,0
14                 BREQ    U_RCV_EXIT      ; То переход на выход
15
16                 CPI     OSRG,'A'        ; Мы готовы. Это 'A'?
17                 BRNE    U_RCV_EXIT      ; Нет? Тогда выход!
18
19                 SetTask TS_OnWhite      ; Зажечь бело-лунный!
20
21 U_RCV_EXIT:     POP     Tmp2
22                 POP     XH              ; Процедура выхода из прерывания
23                 POP     XL
24                 pop     OSRG            ; Восстанавливаем регистры
25                 out     SREG,OSRG
26                 pop     OSRG
27                 RETI                    ; <<<<<< Выходим
28
29 Armed:          LDI     OSRG,1          ; Взводим флаг готовности
30                 STS     R_flag,OSRG     ; Сохраняем его в ОЗУ
31
32 ; Запускаем по таймеру задачу которая сбросит флаг через 10мс
33                 SetTimerTask   TS_ResetR,10
34
35                 RJMP    U_RCV_EXIT      ; Переход к выходу
```

Не смотрите что прерывание такое страшное, просто тут проверка по кучи условий. Также не смущайтесь того, что выход из прерывания не в конце кода, а в самой заднице середине обработчика. Какая, собственно, разница? Никуда она из JMP не выберется. Зато сэкономил на лишнем переходе :) А в коде можно и отбивку сделать :))))) Ну и обратите внимание на то что я сохраняю в стеке. А именно рабочий регистр OSRG, Пару X, и дополнительный Tmp регистр. Забудешь что нибудь из этого — схватишь трудно уловимый глюк который вылезти может только через несколько месяцев агрессивного тестинга.

Что там осталось? Добавить задачи в сетку? Ну тут все просто:

```
1  ;-----------------------------------------------------------------------------
2  OnWhite:        SBI     PORTB,3
3                  SetTimerTask   TS_OffWhite,1000
4                  RET
5  ;-----------------------------------------------------------------------------
6  OffWhite:       CBI     PORTB,3
7                  RET
```

```
8  ;-----------------------------------------------------------------------------
9  ResetR:         CLR     OSRG
10                 STS     R_Flag,OSRG     ; Сбросить флаг готовности
11                 RET
```

Элементарно! Вроде бы ничего не забыл. Задача решена, еще куча ресурсов свободных осталось. Еще без проблем, не затрагивая уже написаное, можно повесить опрос клавиатуры, вывод на индикацию. Скомпильнул, прошил, все с первого раза заработало как надо. А мозг включать даже не пришлось.

Готовый проект под ATMega8, чтобы прошить и посмотреть на это в действии. [1]

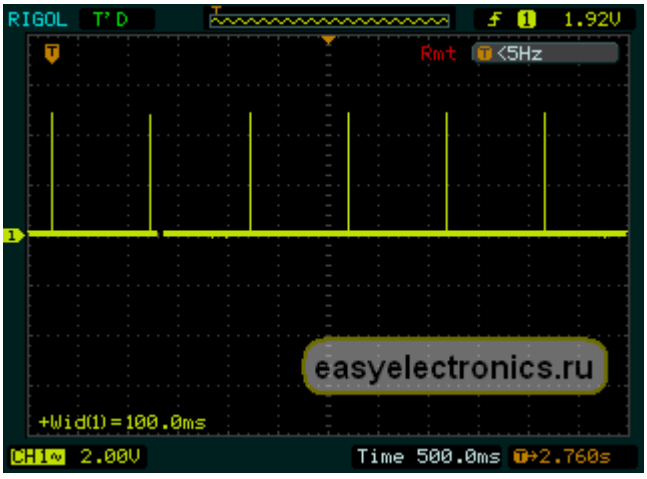
Ну как? Стоило оно того?

AVR. Учебный Курс. Оценка загрузки контроллера.

Как оценить загруженность микроконтроллера? С памятью все понятно — размеры занимаемого кода и оперативной памяти показывает компилятор, а что делать с процессорным временем? Конечно, в линейной программе можно взять и посчитать время выполнения каждой процедуры и станет ясно успеет микроконтроллер выполнить все на него повешанное или слажает в каком-нибудь критичном месте.
Куда сложней оценивать время в кооперативной операционной системе реального времени. Тут задачка получается нетривиальной — у нас куча процессов скачут через диспетчер. В ходе программирования задачи навешиваешь одну за другой, как бусинки на нить — каждый процесс обработки чего либо составляет подобную цепочку, а всего их может быть просто тьма. Ядро же у контроллера всего одно, а значит выполнять можно всего одну задачу за раз и если у нас в диспетчере скопится много критичных ко времени процессов (вообще их лучше развешивать на прерывания, но бывает и прерываний на всех не напасешься), то возможно либо переполнение очереди диспетчера, либо превышение времени ожидания, что тоже не праздник.
Самое западло в том, что умозрительно отлаживать такие вещи довольно сложно. Единственный вариант — рисовать временные диаграммы запуска каждой задачи и смотреть где у нас узкие места. Еще можно попробовать в **AVR Studio** поставить **Break Point** на переполнение диспетчера, но студия не сэмулирует всю ту прорву периферии, а в пошаговой отладке этого не увидеть — да и момент надо подобрать так, чтобы все навалилось.

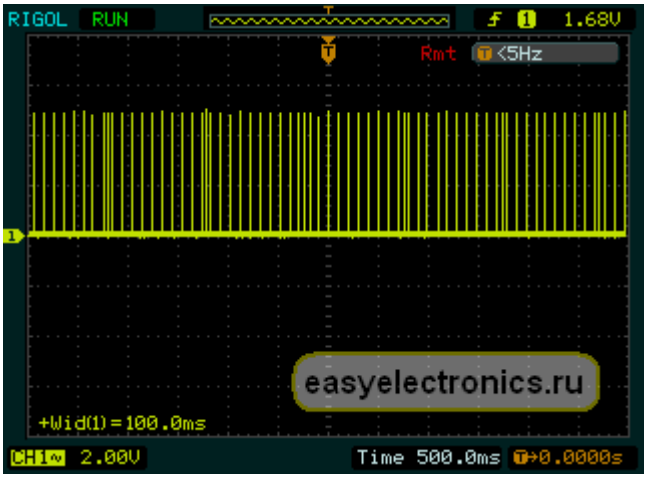
В один момент мне пришла в голову одна идея — а почему бы не заставить рисовать временные диаграммы работы задач сам контроллер? Это же просто! Берем и в диспетчере, перед вызовом задачи выставляем бит порта в 1. А когда диспетчер задач опустошается полностью, то есть выполняется переход на **Idle** — сбрасываем бит в 0. В результате, у нас на выходе будет подобие ШИМ. Если постоянно крутится **Idle** — будут нули перманентно. Если же проц в поте лица гонит через себя непрерывно код, то будут высокий уровнь сплошняком. А если все прерывисто — что то ШИМообразное. Причем чем больше загрузка процессора тем выше заполнение. Можно поставить интегрирующую RC цепочку и получим аналоговый сигнал. Хоть на стрелочный индикатор заводи :).
Сказано — сделано.

Получилось вот так:

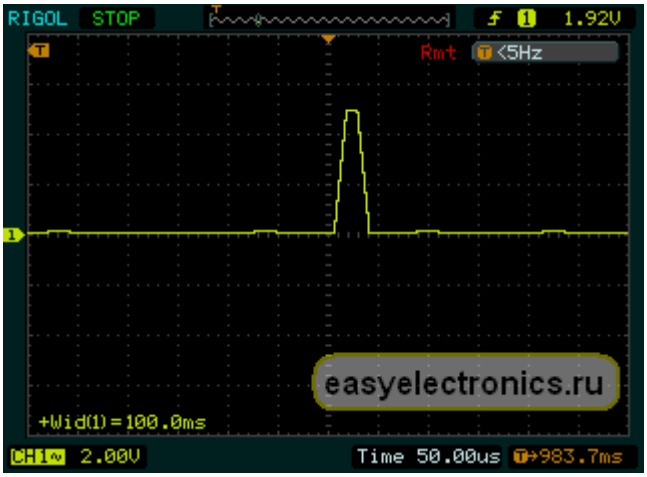


Пока задача шлет раз в секунду байт по UART, а остальное время Idle, т.е. бездельничает.

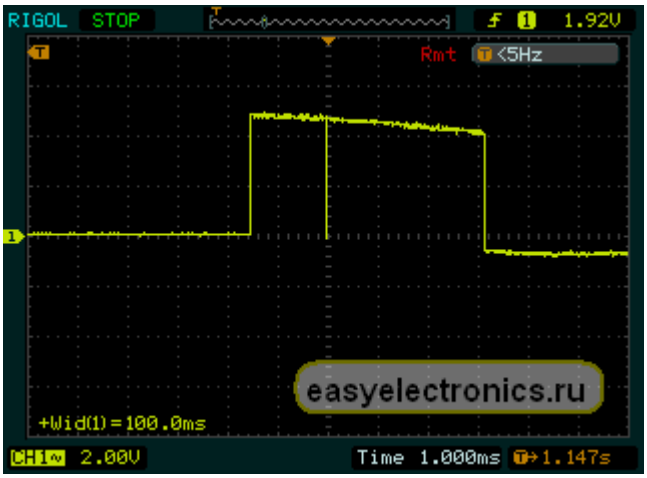
А вот мы добавили опрос клавиатуры. Стало бодрей — иголки увеличились числом. Каждая иголка запуск процесса.



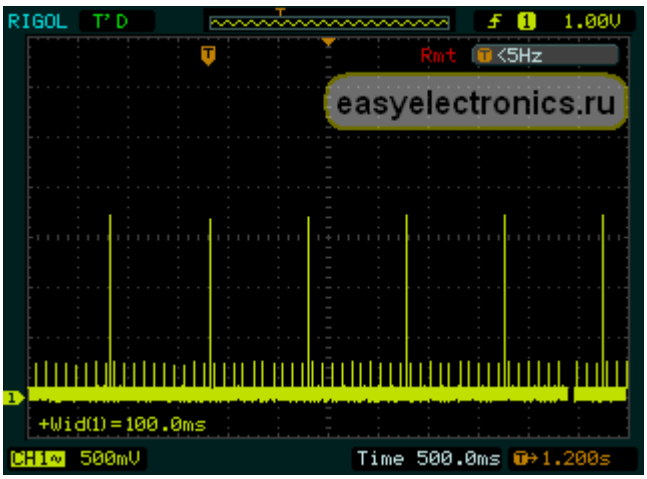
Вот одна иголка крупным планом.



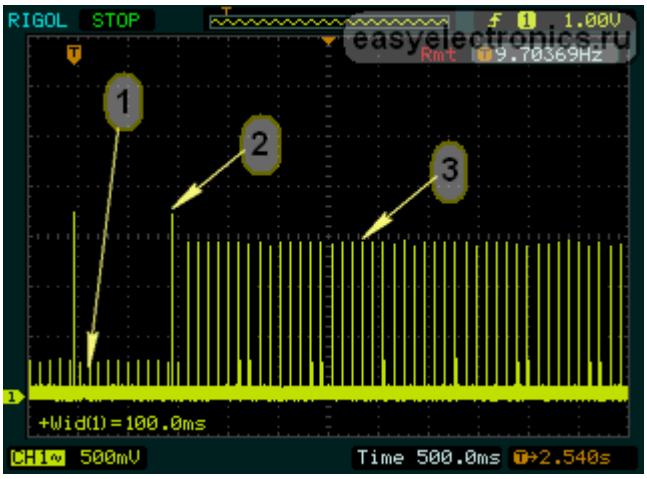
Вызваю процедуру записи в **EEPROM** — во как, сразу же сожралось куча времени на выполнение. А записалось всего 6 байт. Обратите внимание на масштаб времени. Насколько запись в EEPROM дольше выполнения обычного кода.



Но одними иголками сыт не будешь. Не прикольно. Как бы задачки эти выделить. Чтобы можно было понять кто есть кто. Решение элементарное — выделить для отладочных целей не один бит, а поболее. У меня тут в запасе нашлось целых 8 ног. Соответственно я сразу же по заходу в задачу вывел в порт ее номер, а из порта загнал в R-2R ЦАП [1]. Картина стала наглядней — теперь высота иголки у всех задач стала разной.



Чем меньше номер, тем меньше напряжение с ЦАП. Мало того, если задачи вызываются последовательно, а не через очередь таймеров то будут не иголки, а лесенки.



Вот что мы видим. Иголки номер 1 — это сканирование клавиатуры. Те что повыше, номер 2 — это пинг, отсыл одного байта по UART, а вот средние — отправка пачек данных через тот же UART.

Вообще применение осциллографа это могучее средство для реалтаймовой отладки. Не обязательно таким замудреным способом как у меня. Это я больше для прикола и наглядности всякие АЦП вешаю. Достаточно же просто бит выводить и смотреть на осциллографе. Еще можно добавлять программные ловушки. Например, переполнилась критическая переменная — выдали бит. А чтобы не ловить на экране осциллографа этих тараканов, проще этот бит завести на какую-либо следящую систему, да хоть на триггер. Взвел следилку и гоняй свою отлаживаемую программу в хвост и в гриву, нагружай пока не позеленеет от натуги. А если где то что то сорвет, то выскочит твой бит, триггер его запомнит и там хоть сирену включай — ахтунг! Error!!!

Для большей наглядности решил я записать видяшку с происходящим. Там я нагружаю микроконтроллер и показываю как меняется диаграмма загрузки. На некоторых кадрах есть сильная засветка — эт мой светлый лик, заметил уже когда залил на комп, а переснимать мне было влом. Да, думаю, и так все самое интересное видно. Также не смотрите на то, что диаграммы иногда плывут по оси Y — я забыл переключить осцилл на постоянное напряжение.

Однобитные иголки

Нагрузили еще немного. Шлю по UART, видно как растет нагрузка на конвеер

Применили R-2R ЦАП.

AVR. Учебный Курс. Управляемый вектор прерывания

Бывает такая ситуация, когда надо на один периферийный девайс повесить много разных задач, а он всего один и что то надо с этим делать.

Простой пример — таймер и его прерывание по переполнению.
Мы можем задавать выдержку и по прерыванию делать какие-нибудь операции. Но если в один момент времени мы хотим чтобы таймер по прерванию сделал одну операцию, а потом другую, третью. Да сколько угодно, в зависимости от состояния. А вектор один.

Или, например, USART. Нам запросто может потребоваться, чтобы в зависимости от режима на прерывание по приходу байта выполнялся разный код. В одном режиме — выдача приветствия, в другом посыл матом в баню. В третьем удар в голову. А вектор один.

Конечно, можно добавить в обработчик прерывания switch-case конструкцию и по выбору режима перейти на нужный участок кода, но это довольно громоздко, а самое главное — время перехода будет разное, в зависимости от того в каком порядке будет идти опрос-сравнение switch-case структуры.

То есть в свитче вида:

```
1 switch(x)
2         {
3         1: Действие 1
4         2: Действие 2
5         3: Действие 3
6         4: Действие 4
7         }
```

Будет последовательное сравнение x вначале с 1, потом с 2, потом с 3 и так до перебора всех вариантов. А в таком случае реакция на Действие 1 будет быстрей чем реакция на Действие 4. Особо важно это при расчете точных временных интервалов на таймере.

Но есть простое решение этой проблемы — индексный переход. Достаточно перед тем как мы начнем ожидать прерывание предварительно загрузить в переменные (а можно и сразу в индексный регистр Z) направление куда нам надо перенаправить наш вектор и воткнуть в обработчик прерывания индексный переход. И вуаля! Переход будет туда куда нужно, без всякого сравнения вариантов.

В памяти создаем переменные под плавающий вектор:

```
1 Timer0_Vect_L: .byte   1       ; Два байта адреса, старший и младший
2 Timer0_Vect_H: .byte   1
```

Подготовка к ожиданию прерывания проста, мы берем и загружаем в нашу переменную нужным адресом

```
1                CLI                            ; Критическая часть. Прерывания OFF
2                LDI     R16,low(Timer_01)      ; Берем адрес и сохраняем
3                STS     Timer0_Vect_L,R16      ; его в ячейку памяти.
4
5                LDI     R16,High(Timer_01)     ; Аналогично, но уже со старшим вектором
6                STS     Timer0_Vect_H,R16
7                SEI                            ; Прерывания ON
```

Все, можно запускать таймер и ждать нашего прерывания. С другими случаями аналогично.

А обработчик получается вида:

```
1  ;============================
2  ; Вход в прерывание по переполнению от Timer0
3  ;============================
4  TIMER_0:        PUSH    ZL              ; сохраняем индексный регистр в стек
5                  PUSH    ZH              ; т.к. мы его используем
6                  PUSH    R2              ; сохраняем R2, т.к. мы его тоже портим
```

```
7                IN      R2,SREG         ; Извлекем и сохраняем флаговый регистр
8                PUSH    R2              ; Если не сделать это, то 100% получим глюки
9
10               LDS     ZL,Timer0_Vect_L              ; загружаем адрес нового вектора
11               LDS     ZH,Timer0_Vect_H              ; оба байта.
12
13               CLR     R2              ; Очищаем R2
14               OR      R2,ZL           ; Проверяем вектор на ноль. Иначе схватим аналог
15               OR      R2,ZH           ; reset'a. Проверка идет через операцию OR
16               BREQ    Exit_Tm0        ; с накоплением результата в R2
17                                       ; так мы не портим содержимое Z и нам не придется
18                                       ; загружать его снова
19               IJMP                    ; Уходим по новому вектору
20
21 ; Выход из прерывания.
22 Exit_Tm0:     POP     R2              ; Достаем и восстанавливаем регистр флагов
23               OUT     SREG,R2
24               POP     R2              ; восстанавливаем R2
25               POP     ZH              ; Восстанавливаем Z
26               POP     ZL
27               RETI
28
29 ; Дополнительный вектор 1
30 Timer_01:     NOP                     ; Это наши новые вектора
31               NOP                     ; тут мы можем творить что угодно
32               NOP                     ; желательно недолго - в прерывании же
33               NOP                     ; как никак. Если используем какие другие
34               NOP                     ; регистры, то их тоже в стеке сохраняем
35               RJMP    Exit_Tm0        ; Это переход на выход из прерывания
36                                       ; специально сделал через RJMP чтобы
37 ; Дополнительный вектор 2             ; сэкономить десяток байт на коде возврата :)))
38 Timer_02:     NOP
39               NOP
40               NOP
41               NOP
42               NOP
43               RJMP    Exit_Tm0
44 ; Дополнительный вектор 3
45 Timer_03:     NOP
46               NOP
47               NOP
48               NOP
49               NOP
50               RJMP    Exit_Tm0
```

Реализация для RTOS

Но что делать если у нас программа построена так, что весь код вращается по цепочкам задач через диспетчер RTOS? Просчитать в уме как эти цепочки выполняются относительно друг друга очень сложно. И каждая из них может попытаться завладеть таймером (конечно не самовольно, с нашей подачи, мы же программу пишем, но отследить по времени как все будет сложно).

В современных больших осях на этот случай есть механизм Mutual exclusion — mutex. Т.е. это своего рода флаг занятости. Если какой нибудь процесс общается, например, с UART то другой процесс туда байт сунуть не смеет и покорно ждет пока первый процесс освободит UART, о чем просемафорит флажок.

В моей RTOS [1] механизмов взаимоисключений нет, но их можно реализовать. По крайней мере сделать некоторое минимальное подобие. Полноценную реализацию всего этого барахла я делать не хочу, т.к. моей целью является удержания размера ядра на уровне 500-800 байт.

Проще всего зарезервировать в памяти еще один байт — переменную занятости. И когда один процесс захватывает ресурс, то в эту переменную он записывает время когда ориентировочно он его освободит. Время идет в тиках системного таймера которое у меня 1ms.

Если какой либо другой процесс попытается обратиться к этому же аппаратному ресурсу, то он вначале посмотрит на состояние его занятости, считает время в течении которого будет занято и уйдет покурить на этот период — загрузит сам себя в очередь по таймеру. Там снова проверит и так далее. Это простейший вариант.

Проблема тут в том, что если на один вектор много желающих будет, то процессы так и будут бегать вокруг да около, словно бухая молодежь вокруг единственного сортира на площади в период праздничных гуляний. У кого нибудь да мочевой пузырь не выдержит — запорет алгоритм. А у кого тут фиг угадаешь, т.к. промоделировать это будет сложновато.

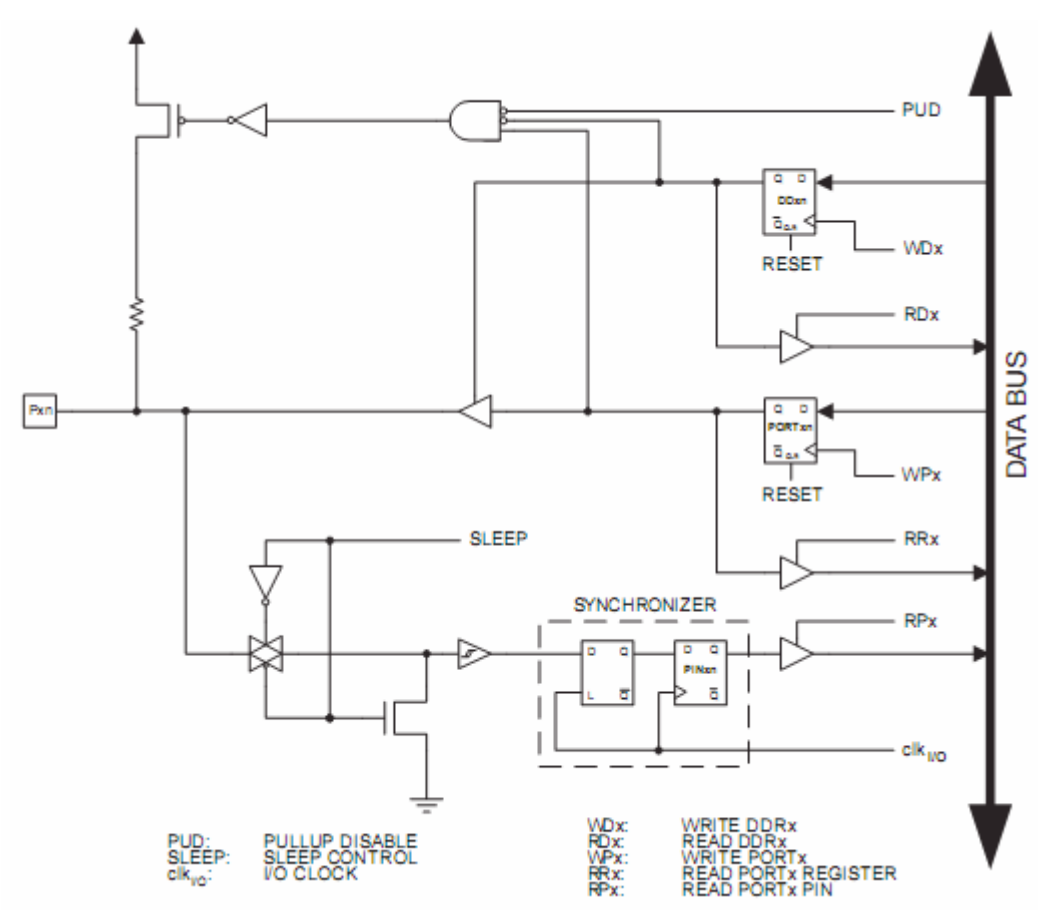
Решение проблемы — добавление еще одной очередной цепочки, на этот раз уже на доступ к ресурсу. Чтобы он не простаивал вообще. Т.е. один выскочил, тут же второй, третий и так далее пока все процессы не справят свою нужду в какой нибудь там USART.
Недостаток очевиден — еще одна очередь это дополнительная память, дополнительный код, дополнительное время. Можно, конечно, извратиться и на очередь к вектору натравить код диспетчера основной цепи. Но тут надо все внимательно отлаживать, ведь вызываться он будет по прерыванию! Да и громоздко, требуется лишь тогда, когда у нас много желающих.

Второе решение — выкинуть переменную времени занятости, оставив только флаг «Занято!». А процесс который пытается обратиться не убегает покурить, а отскакивает на пару шагов назад — на конец очереди задач и сразу же ломится обратно. Народ вокруг сортира не вокруг бегает, а толкется локтями у входа по принципу кто первый пролезет.
Недостаток другой — большая нагрузка на главный конвеер, куча запросов на постановку в очередь так недолго распухнуть на всю оперативку и повстречаться со стеком, а это черевато глобальным апокалипсисом.

Разумеется таймер тут приведен для примера, большую часть задач можно решить системным таймером RTOS, но если нужна вдруг меньшая дискретность или высокая скорость реакции на событие (а не пока главный конвеер дотащит задачу до исполнения), то механим управляемых прерываний, ИМХО, то что доктор прописал.

AVR. Учебный курс. Устройство и работа портов ввода-вывода

С внешним миром микроконтроллер общается через порты ввода вывода. Схема порта ввода вывода указана в даташите:



Но новичку там разобраться довольно сложно. Поэтому я ее несколько упростил:

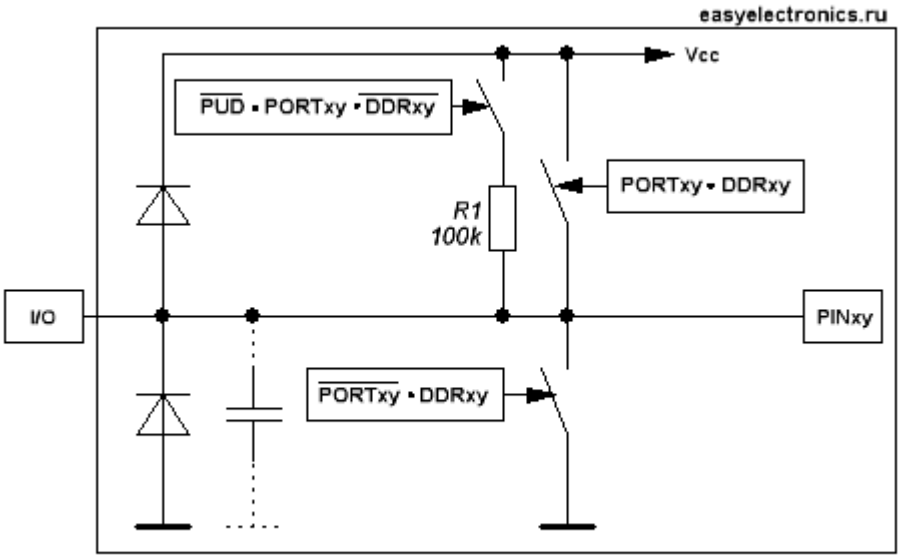


Схема одного вывода порта МК AVR без учета доп функций

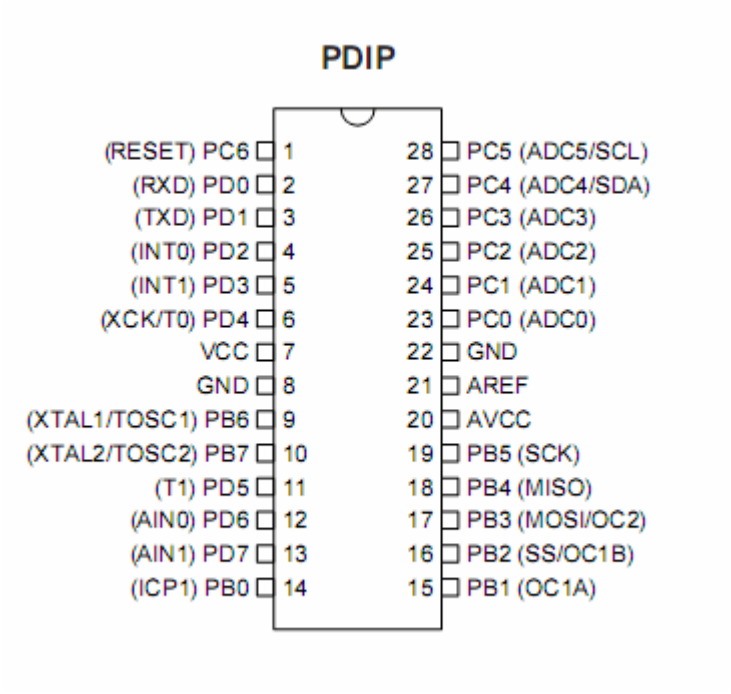
Итак, что же представляет собой один вывод микроконтроллера. Вначале на входе стоит небольшая защита из диодов, она призвана защитить ввод микроконтроллера от превышения напряжения. Если напряжение будет выше питания, то верхний диод откроется и это напряжение будет стравлено на шину питания, где с ним будет уже бороться источник питания и его фильтры. Если на ввод попадет отрицательное (ниже нулевого уровня) напряжение, то оно будет нейтрализовано через нижний диод и погасится на землю. Впрочем, **диоды там хилые и защита эта помогает только от микроскопических импульсов от помех**. Если же ты по ошибке вкачаешь в ножку микроконтроллера вольт 6-7 при 5 вольтах питания, то никакой диод его не спасет.

Конденсатор, нарисованный пунктиром, это паразитная емкость вывода. Хоть она и крошечная, но присутствует. Обычно ее не учитывают, но она есть. Не забивай голову, просто знай это, как нибудь я тебе даже покажу как её можно применить ;)

Дальше идут ключи управления. Это я их нарисовал рубильниками, на самом деле там стоят полевые транзисторы, но особой сути это не меняет. А рубильники наглядней.
Каждый рубильник подчинен логическому условию которое я подписал на рисунке. Когда условие выполняется — ключ замыкается. **PIN, PORT, DDR** это регистры конфигурации порта.

Есть в каждом контроллере **AVR** (в **PIC** есть тоже подобные регистры, только звать их по другому).

Например, смотри в даташите на цоколевку микросхемы:



Видишь у каждой почти ножки есть обозначение **Pxx**. Например, **PB4**где буква **«B»** означает имя порта, а цифра — номер бита в порту. За порт «B» отвечают три восьмиразрядных регистра **PORTB, PINB, DDRB**, а каждый бит в этом регистре отвечает за соответствующую ножку порта. За порт «**A**» таким же образом отвечают **PORTA, DDRA, PINA**.

PINx
Это регистр чтения. Из него можно только читать. В регистре **PINx** содержится информация о **реальном текущем логическом уровне** на выводах порта. Вне зависимости от настроек порта. Так что если хотим узнать что у нас на входе — читаем соответствующий бит регистра **PINx**Причем существует две границы: граница гарантированного нуля и граница гарантированной единицы — пороги за которыми мы можем однозначно четко определить текущий логический уровень. Для пятивольтового питания это 1.4 и 1.8 вольт соответственно. То есть при снижении напряжения от максимума до минимума бит в регистре PIN переключится с 1 на 0 только при снижении напруги ниже 1.4 вольт, а вот когда напруга нарастает от минимума до максимума переключение бита с 0 на 1 будет только по достижении напряжения в 1.8 вольта. То есть возникает гистерезис переключения с 0 на 1, что исключает хаотичные переключения под действием помех и наводок, а также исключает ошибочное считывание логического уровня между порогами переключения.

При снижении напряжения питания разумеется эти пороги также снижаются, график зависимости порогов переключения от питающего напряжения можно найти в даташите.

DDRx
Это регистр направления порта. Порт в конкретный момент времени может быть либо входом либо выходом (но для состояния битов **PIN** это значения не имеет. Читать из PIN реальное значение можно всегда).

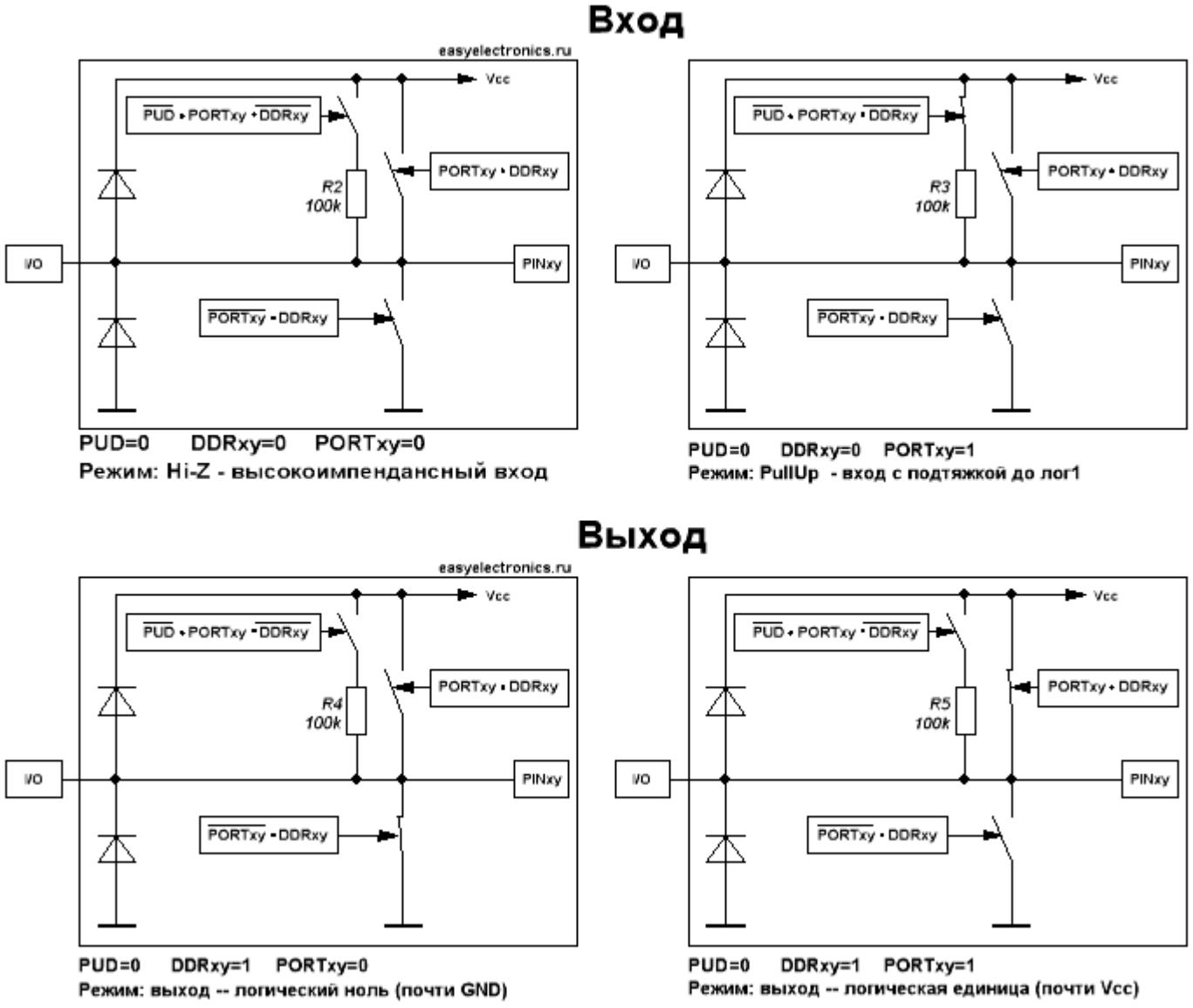
- DDRxy=0 — вывод работает как ВХОД.
- DDRxy=1 вывод работает на ВЫХОД.

PORTx
Режим управления состоянием вывода. Когда мы настраиваем вывод на вход, то от **PORT** зависит тип входа (Hi-Z или PullUp, об этом чуть ниже).
Когда ножка настроена на **выход**, то значение соответствующего бита в регистре PORTx определяет состояние вывода. Если **PORTxy=1** то на выводе лог1, если **PORTxy=0** то на выводе лог0.
Когда ножка настроена на **вход**, то если **PORTxy=0**, то вывод в режиме **Hi-Z**. Если **PORTxy=1** то вывод в режиме **PullUp** с подтяжкой резистором в 100к до питания.

Есть еще бит **PUD** (PullUp Disable) в регистре **SFIOR** он запрещает включение подтяжки сразу для всех портов. По дефолту он равен 0. Честно говоря, я даже не знаю нафиг он нужен — ни разу не доводилось его применять и даже не представляю себе ситуацию когда бы мне надо было запретить использование подтяжки сразу для всех портов. Ну да ладно, инженерам **Atmel** видней, просто знай что такой бит есть. Мало ли, вдруг будешь чужую

прошивку ковырять и увидишь что у тебя подтяжка не работает, а вроде как должна. Тогда слазаешь и проверишь этот бит, вдруг автор прошивки заранее где то его сбросил.

Общая картина работы порта показана на рисунке:



Теперь кратко о режимах:

- **Режим выхода**
  Ну тут, думаю, все понятно — если нам надо выдать в порт 1 мы включаем порт на выход (**DDRxy=1**) и записываем в **PORTxy** единицу — при этом замыкается верхний ключ и на выводе появляется напряжение близкое к питанию. А если надо ноль, то в **PORTxy** записываем 0 и открывается уже нижний вентиль, что дает на выводе около нуля вольт.
- **Вход Hi-Z** — режим высокоимпендансного входа.
  Этот режим включен по умолчанию. Все вентили разомкнуты, а сопротивление порта **очень велико**. В принципе, по сравнению с другими режимами, можно его считать бесконечностью. То есть электрически вывод как бы вообще никуда не подключен и ни на что не влияет. Но! При этом он постоянно считывает свое состояние в регистр **PIN** и мы всегда можем узнать что у нас на входе — единица или ноль. Этот режим хорош для прослушивания какой либо шины данных, т.к. он не оказывает на шину никакого влияния. А что будет если вход висит в воздухе? А в этом случае напряжение будет на нем скакать в зависимости от внешних наводок, электромагнитных помех и вообще от фазы луны и погоды на Марсе

(идеальный способ нарубить случайных чисел!). Очень часто на порту в этом случае нестабильный синус 50Гц — наводка от сети 220В, а в регистре PIN будет меняться 0 и 1 с частотой около 50Гц

- **Вход PullUp** — вход с подтяжкой.
  При **DDRxy=0** и **PORTxy=1** замыкается ключ подтяжки и к линии подключается резистор в 100кОм, что моментально приводит неподключенную никуда линию в состояние лог1. Цель подтяжки очевидна — недопустить хаотичного изменения состояния на входе под действием наводок. Но если на входе появится логический ноль (замыкание линии на землю кнопкой или другим микроконтроллером/микросхемой), то слабый 100кОмный резистор не сможет удерживать напряжение на линии на уровне лог1 и на входе будет нуль.

Также почти каждая ножка имеет **дополнительные функции**. На распиновке они подписаны в скобках. Это могут быть выводы приемопередатчиков, разные последовательные интерфейсы, аналоговые входы, выходы ШИМ генераторов. Да чего там только нет. По умолчанию **все эти функции отключены**, а вывод управляется исключительно парой **DDR** и **PORT**, но если включить какую-либо дополнительную функцию, то тут уже управление может полностью или частично перейти под контроль периферийного устройства и тогда хоть запишись в **DDR/PORT** — ничего не изменится. До тех пор пока не выключишь периферию занимающую эти выводы.
Например, приемник **USART**. Стоит только выставить бит разрешения приема **RXEN** как вывод **RxD**, как бы он ни был настроен до этого, переходит в режим входа.

Совет:
С целью снижения энергопотребления и повышения надежности **рекомендуется все неиспользованные пины включить в режим PullUp** тогда их не будет дергать туда сюда помехой, а если на порт свалится грубая сила (например, монтажник отвертку уронит и коротнет на землю) то линия не выгорит.

Как запомнить режимы, чтобы не лазать каждый раз в справочник:
Чем зазубривать или писать напоминалки, лучше понять логику разработчиков, проектировавших эти настройки, и тогда все запомнится само.

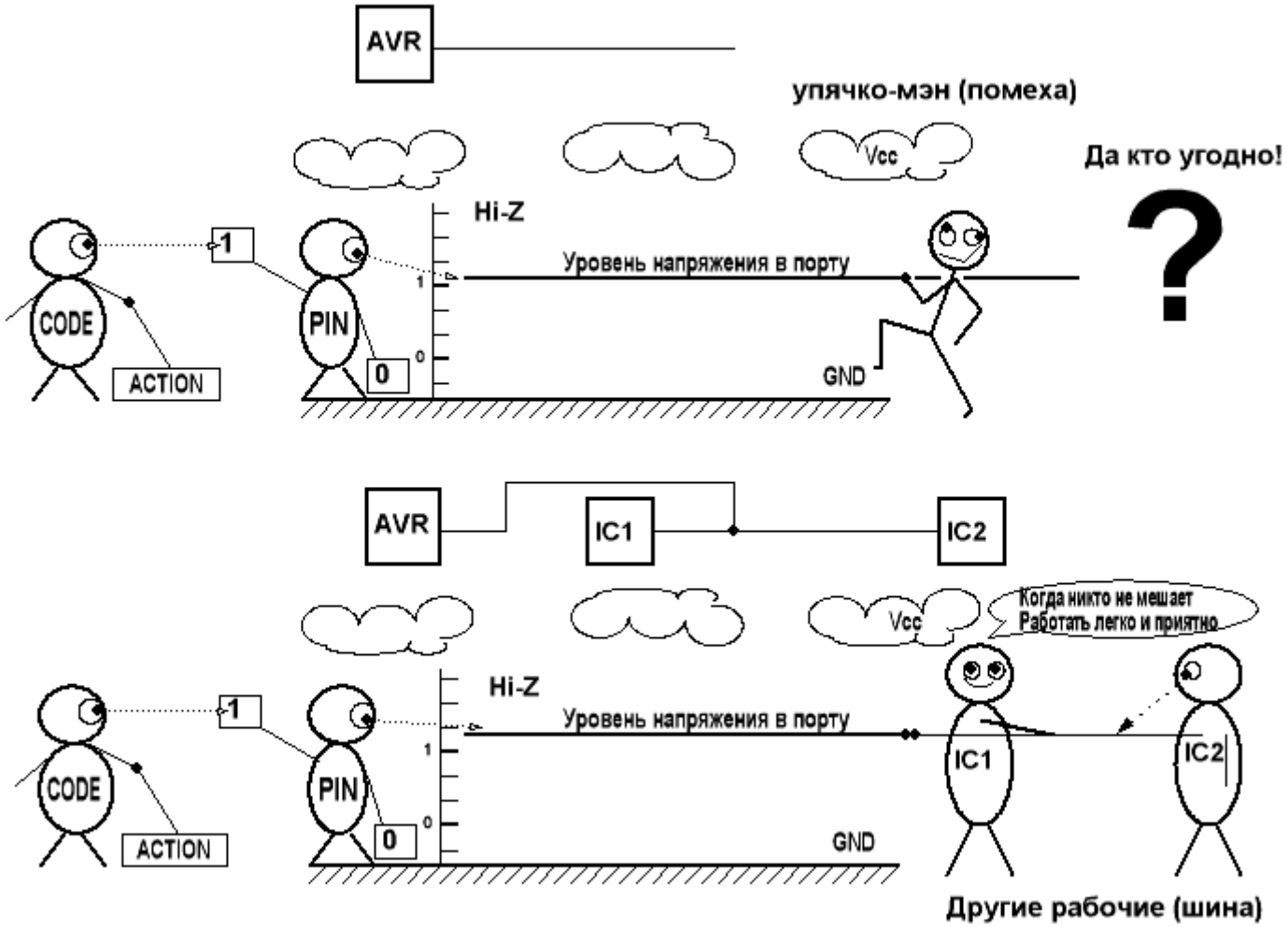
Итак:

- Самый безопасный для МК и схемы, ни на что не влияющий режим это Hi-Z.
- Очевидно что этот режим и должен быть по дефолту.
- Значения большинства портов I/O при включении питания/сбросе = 0x00, PORT и DDR не исключение.
- Соответственно когда DDR=0 и PORT=0 это High-Z — самый безопасный режим, оптимальный при старте.
- Hi-Z это вход, значит при DDR=0 нога настроена на вход. Запомнили.
- Однако, если DDR=0 — вход, то что будет если PORT переключить в 1?
- Очевидно, что будет другой режим входа. Какой? Pullup, другого не дано! Логично? Логично. Запомнили.
- Раз дефолтный режим был входом и одновременно в регистрах нуль, то для того, чтобы настроить вывод на **выход** надо в DDR записать 1.
- Ну, а состояние выхода уже соответствует регистру PORT — высокий это 1, низкий это 0.
- Читаем же из регистра PIN.

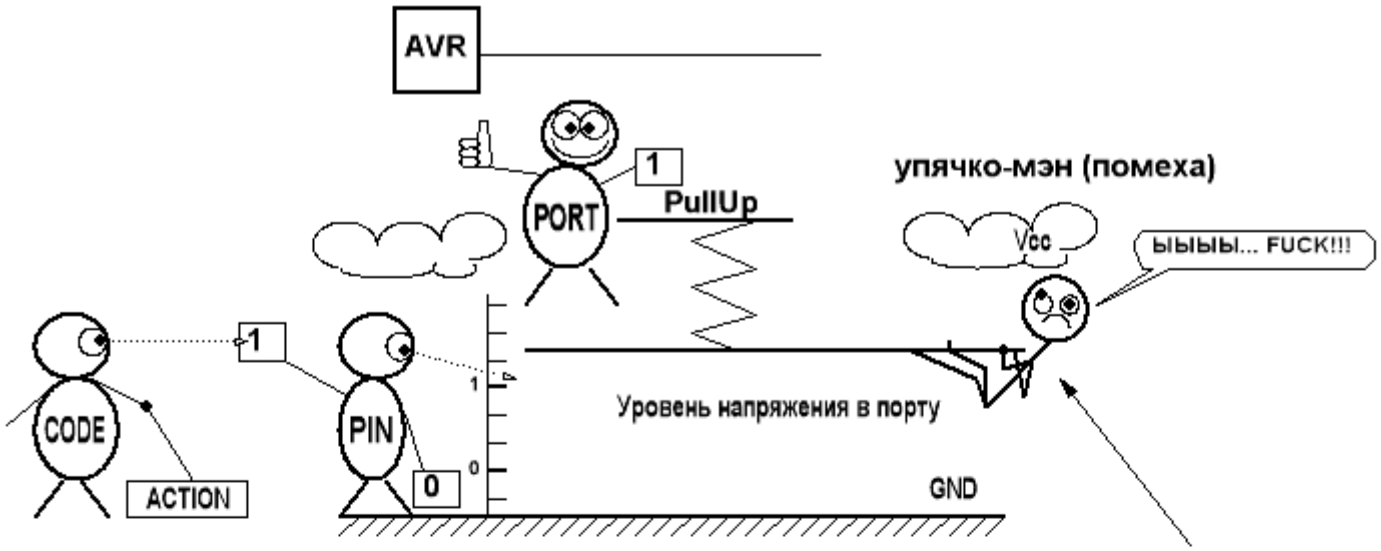
Есть еще один способ, мнемонический:
1 похожа на стрелку. Стрелка выходящая из МК — выход. Значит DDR=1 это выход! 0 похож на гнездо, дырку — вход! Резистор подтяжки дает в висящем порту единичку, значит PORT в режиме Pullup должен быть в единичке!

Все просто! :)

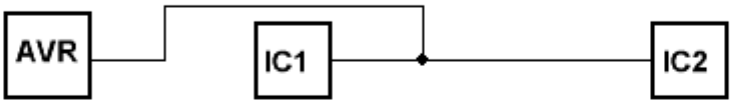
Для детей в картинках и комиксах :)
Для большей ясности с режимами приведу образный пример:

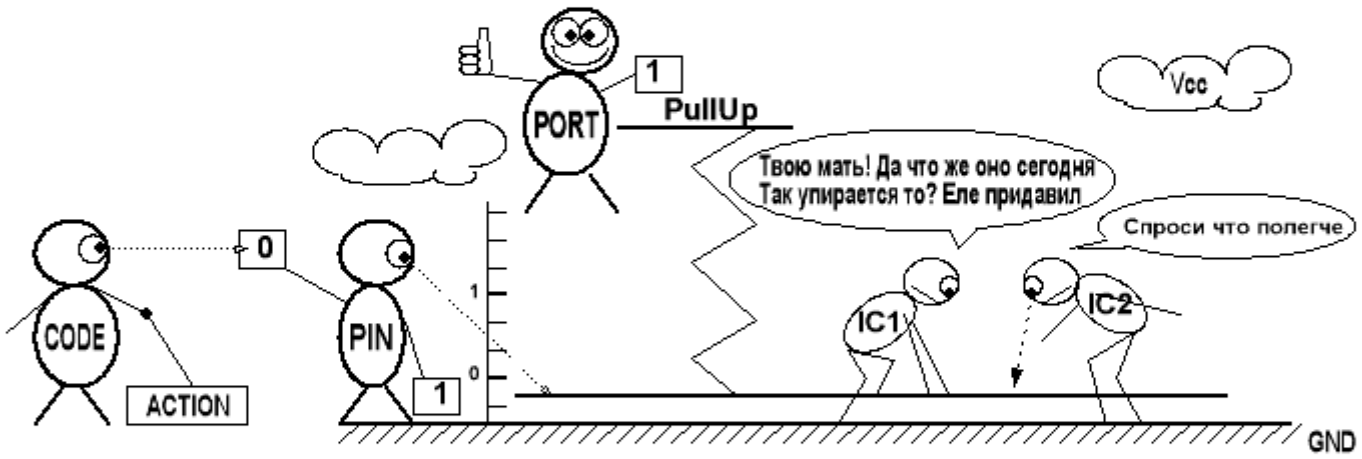


Уровень напряжения на выводе словно планка, которая может двигаться вертикально вверх или вниз. В режиме **Hi-Z** мы можем на эту планку только смотреть, а двигать или как то на нее воздействовать мы не можем. Поэтому любая помеха может ее дрыгать как угодно, но зато если мы ее куда прицепим, то ее уровень будет зависеть только от другой цепи и ей мы не помешаем.

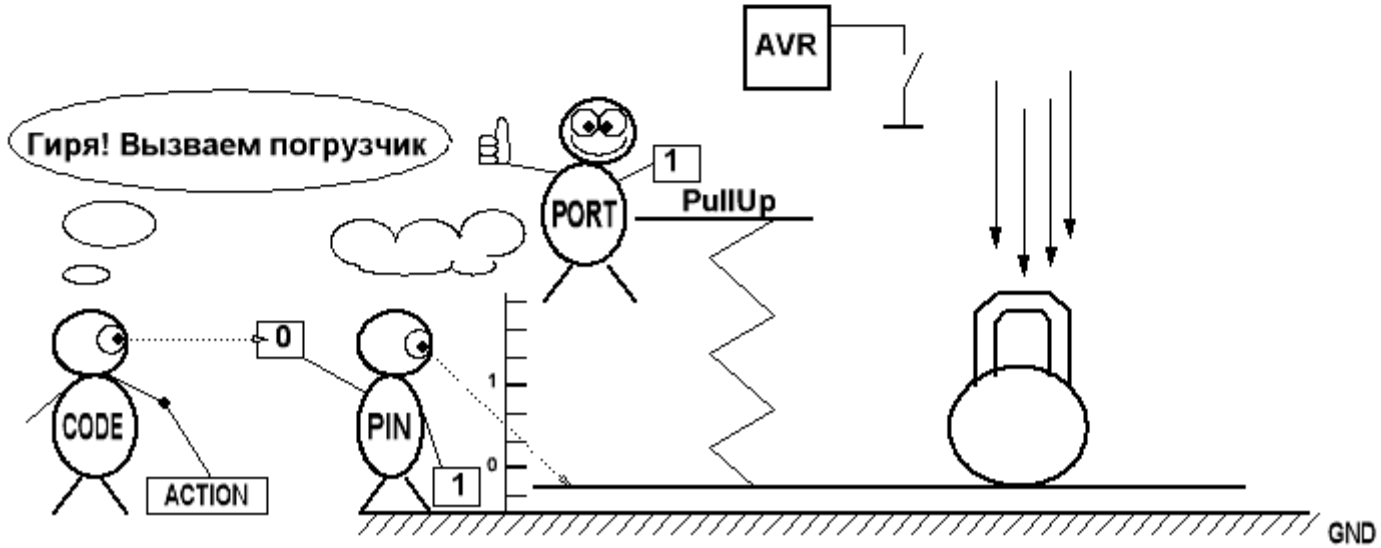


Помехе не хватает сил, чтобы пересилить подтяжку



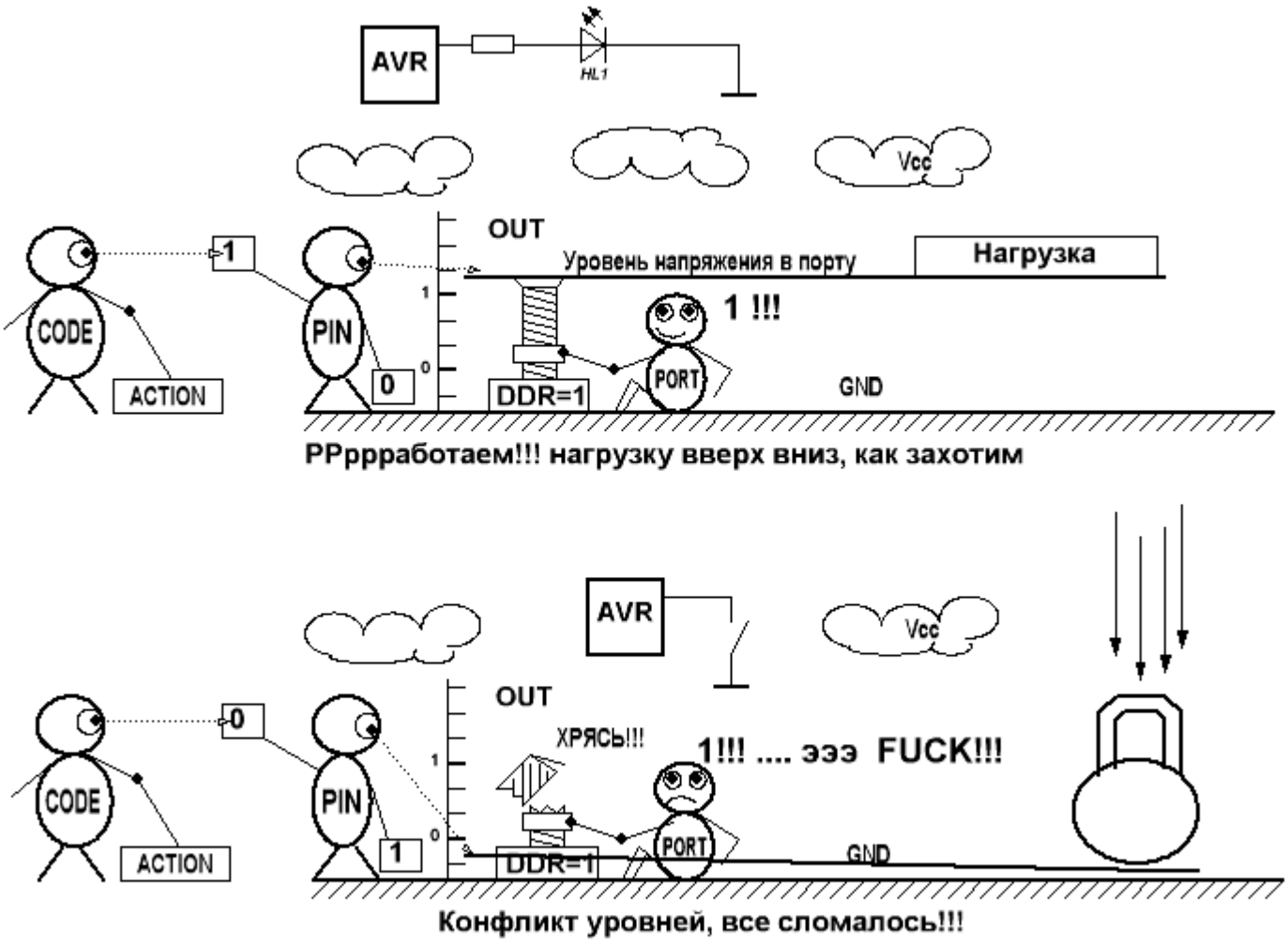


Шина может и работает (если силы хватит), но ее это явно напрягает



А гире все равно!

В режиме **PullUp** эту планку мы пружиной подтянули кверху. Слабые помехи не смогут больше ее дрыгать как угодно. С другой стороны шине она может помешать, но не факт что заблокирует ее работу. От шины зависит и ее силы. А еще мы можем отслеживать тупую внешнюю силу, вроде кнопки, которая может взять и придавить ее к земле. Тогда мы узнаем что кнопка нажата.



В режиме **OUT** у нас планка прибита гвоздями к земле или прижата домкратом к питанию. Внешняя сила может ее пересилить только сломав домкрат или сломается сама. Тупая внешняя сила просто разрушает наш домкрат или вырывает гвозди из пола с мясом. В любом случае — девайс в помойку.

Подключение микроконтроллера. Ликбез.

Казалось бы простая тема, а однако в комментах меня завалили вопросами как подключить микроконтроллер. Как подключить к нему светодиод, кнопку, питание. Что делать с **AGND** или **AREF**. Зачем нужен **AVCC** и все в таком духе. Итак, раз есть вопросы, значит тема не понятна и надо дать по возможности исчерпывающий ответ. Все описываю для контроллеров AVR, но для каких нибудь PIC все очень и очень похоже. Т.к. принципы тут едины.

Чтобы понимать ряд терминов активно упоминающихся в этой статье, надо сначала прочитать статью про порты ввода-вывода. [1]

Питание
Для работы микроконтроллеру нужна энергия — электричество. Для этого на него естественно нужно завести питалово. Напряжение питание у МК **Atmel AVR** разнится от **1.8** до **5** вольт, в зависимости от серии и модели. Все **AVR** могут работать от 5 вольт (если есть чисто низковольтные серии, то просьба уточнить в комментах, т.к. я таких не встречал). Так что будем считать что напряжение питания контроллера у нас всегда 5 вольт или около

того. Плюс напряжения питания обычно обозначается как **Vcc**. Нулевой вывод (а также Земля, Корпус, да как только его не называют) обозначают **GND**. Если взять за пример комповый блок питания. То черный провод это GND (кстати, земляной провод традиционно окрашивают в черный цвет), а красный это +5, будет нашим **Vcc**. Если ты собираешься запитать микроконтроллер от батареек, то минус батареек примем за **GND**, а плюс за **Vcc** (главное чтобы напряжение питания с батарей было в заданных пределах для данного МК, позырь в даташите. Параметр обычно написан на первой странице в общем описании фич:

• Operating Voltages
–1.8 — 5.5V (ATtiny2313V)
–2.7 — 5.5V (ATtiny2313)
• Speed Grades
–ATtiny2313V: 0 — 4 MHz @ 1.8 — 5.5V, 0 — 10 MHz @ 2.7 — 5.5V
–ATtiny2313: 0 — 10 MHz @ 2.7 — 5.5V, 0 — 20 MHz @ 4.5 — 5.5V

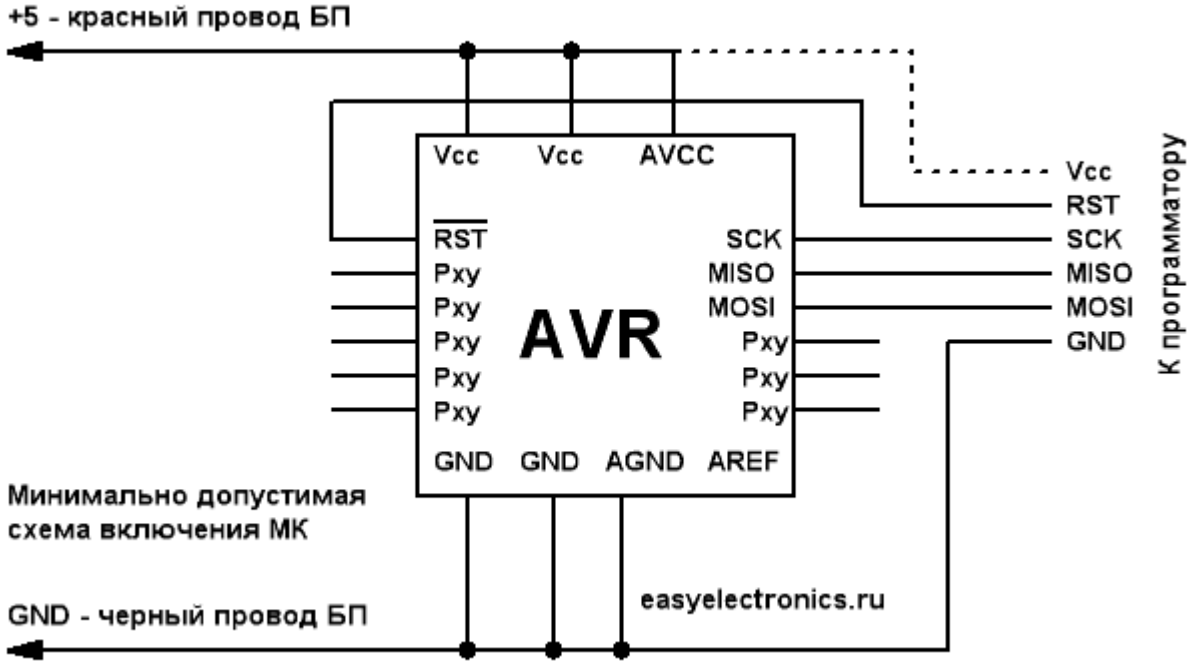
Обрати внимание, что есть особые низковольтные серии (например 2313V низковльтная) у которых нижня граница напряжения питания сильно меньше. Также стоит обратить внимание на следующий пункт, про частоты. Тут показана зависимость максимальной частоты от напряжения питания. Видно, что на низком напряжении предельные частоты ниже. А низковольтные серии раза в два медленней своих высоковольтных коллег. Впрочем, разгону все процессоры покорны ;)))))

Для работы контроллерам серии **AVR** достаточно только питания. На все входы **Vcc** надо подать наши 5 (или сколько там у тебя) вольт, а все входы **GND** надо посадить на землю. У микроконтроллера может быть много входов **Vcc** и много входов **GND** (особенно если он в квадратном **TQFP** корпусе. У которого питалово со всех сторон торчит). Много выводов сделано не для удобства монтажа, а с целью равномерной запитки кристалла со всех сторон, чтобы внутренние цепи питания не перегружались. А то представь, что подключил ты питалово только с одной стороны, а с другой стороны чипа навесил на каждую линию порта по светодиоду, да разом их зажег. Внутренняя тонкопленочная шина питания, офигев от такой токовой нагрузки, испарилась и проц взял ВНЕЗАПНО и без видимых, казалось бы, причин отбросил копыта. Так что **ПОДКЛЮЧАТЬ НАДО ВСЕ ВЫВОДЫ Vcc и GND**. Соединить их соответственно и запитать.

Отдельные вопросы вызвают **AGND** и **AVCC** — это аналоговая земля и питание для Аналого-Цифрового Преобразователя. АЦП это очень точный измеритель напряжения, поэтому его желательно запитать через дополнительные фильтры, чтобы помехи, которые не редки в обычной питающей цепи, не влияли на качество измерения. С этой целью в точных схемах проводят разделение земли на цифровую и аналоговую (они соединены должны быть только в одной точке), а на **AVCC** подается напряжение через фильтрующий дроссель. Если ты не планируешь использовать АЦП или не собираешься делать точные измерения, то вполне допустимо на **AVCC** подать те же 5 вольт, что и на **Vcc**, а **AGND** посадить на ту же землю что и все. **Но подключать их надо обязательно!!!** ЕМНИП от AVCC питается также порт A.

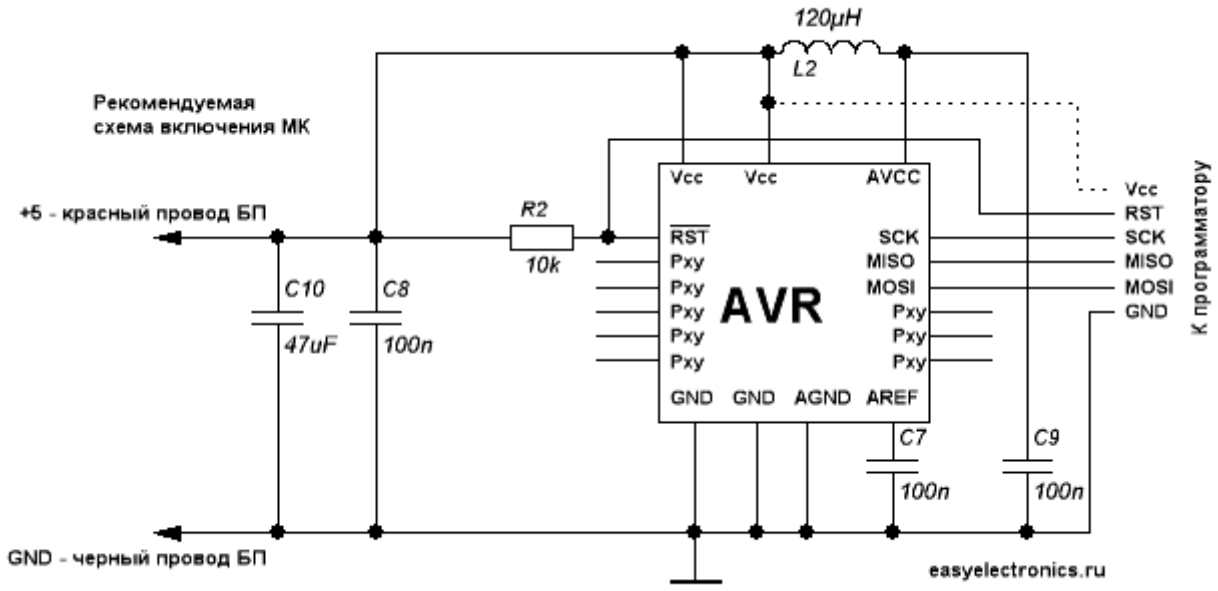
Warning!!!
В чипе Mega8 похоже есть ошибка на уровне топологии чипа — Vcc и AVcc связаны между собой внутри кристалла. Между ними сопротивление около (!!!) 5Ом Для сравнения, в ATmega16 и ATmega168 между Vcc и AVcc сопротивление в десятки МЕГА ом! В даташите на этот счет никаких указаний нет до сих пор, но в одном из топиков за 2004 год на AVRFreaks [2] сказано, что люди бодались с цифровым шумом АЦП, потом написали в поддержку Atmel мол WTF??? А те, дескать, да в чипе есть бага и Vcc и AVcc соединены внутри кристалла. В свете этой инфы, думаю что ставить дроссель на AVcc для Mega8 практически бесполезно. Но AVcc запитывать надо в любом случае — кто знает насколько мощная эта внутренняя связь?

Простейшая схема подключения Микроконтроллера AVR приведена ниже:



Это необходимый минимум чтобы контроллер запустился. Провод **Vcc** до программатора показан пунктиром поскольку он не обязателен. Если ты собираешься питать МК от внешнего источника, то он не нужен. Но я все же рекомендую для начала питать всю систему (МК+программатор) от одного источника — больше вероятность успешной прошивки :) Для учебной цели, диодиком помигать, сойдет и так.

Но настолько все упрощать я не рекомендую. Лучше сразу добавить парочку навесных внешних элементов. Правильней будет. Чтобы было вот так:



Как видишь, добавился дроссель в цепь питания **AVCC**, а также конденсаторы. Хорошим тоном является ставить керамический конденсатор на сотню нанофарад между **Vcc** и **GND** у каждой микросхемы (а если у микрухи много вход питания и земель, то между каждым питанием и каждой землей) как можно ближе к выводам питания — он сгладит краткие импульсные помехи в шине питания вызыванные работой цифровых схем. Конденсатор на 47мКФ

в цепи питания сгладит более глубокие броски напряжения. Кондесатор между **AVcc** и **GND** дополнительно успокоит питание на **АЦП**.

Вход **AREF** это вход опорного напряжения **АЦП**. Туда вообще можно подать напряжение относительно которого будет считать **АЦП**, но обычно используется либо внутренний источник опорного напряжения на 2.56 вольта, либо напряжение на **AVCC**, поэтому на **AREF** рекомендуется вешать конденсатор, что немного улучшит качество опорного напряжения **АЦП** (а от качества опоры зависит адекватность показаний на выходе **АЦП**).

Схема сброса
Резистор на **RESET**. Вообще в **AVR** есть своя внутренняя схема сброса, а сигнал **RESET** изнутри уже подтянут резистором в 100кОм к **Vcc**. НО! Подтяжка это настолько дохлая, что микроконтроллер ловит сброс от каждого чиха. Например, от касания пальцем ножки **RST**, а то и просто от задевания пальцем за плату. Поэтому крайне рекомендуется **RST** подтянуть до питания резистором в 10к. Меньше не стоит, т.к. тогда есть вероятность, что внутрисхемный программатор не сможет эту подтяжку пересилить и прошить МК внутри схемы не удасться. 10к в самый раз.

Есть еще вот такая схема сброса:

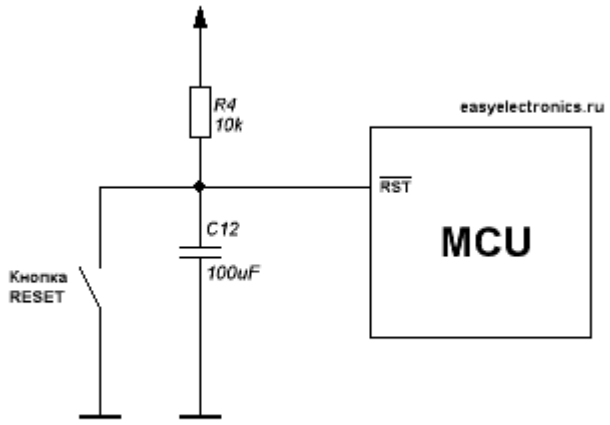
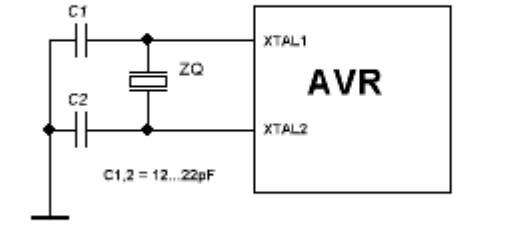


Схема надежного сброса МК

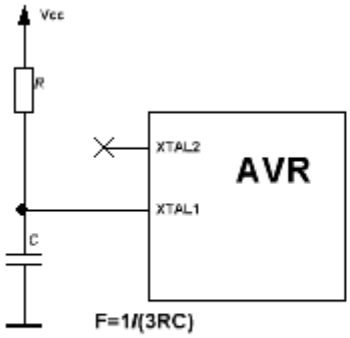
Она замечательна чем — при включении схемы конденсатор разряжен и напряжение на **RST** близко к нулю — микроконтроллер не стартует, т.к. ему непрерывный сброс. Но со временем, через резистор, конденсатор зарядится и напряжение на **RST** достигнет лог1 — МК запустится. Ну, а кнопка позволяет принудительно сделать сброс если надо.
Задержка будет примерно T=R*C для данного примера — около секунды. Зачем эта задержка? Да хотя бы для того, чтобы МК не стартовал раньше чем все девайсы платы запитаются и выйдут на установившийся режим. В старых МК (**AT89C51**, например) без такой цепочки, обеспечивающей начальный сброс, МК мог вообще не стартануть.
В принципе, в **AVR** задержку старта, если нужно, можно сделать программно — потупить с пол секунды прежде чем приступать к активным действиям. Так что кондер можно выкинуть нафиг. А кнопку… как хочешь. Нужен тебе внешний **RESET**? Тогда оставь. Я обычно оставляю.

Источник тактового сигнала
Тактовый генератор это сердце микроконтроллера. По каждому импульсу происходит какая нибудь операция внутри контроллера — гоняют данные по регистрам и шинам, переключаются выводы портов, щелкают таймеры. Чем быстрей тактовая частота тем шустрей МК выполняет свои действия и больше жрет энергии (на переключения логических вентилей нужна энергия, чем чаще они переключаются тем больше энергии надо).

Импульсы задаются тактовым генератором встроенным в микроконтроллер. Впрочем может быть и внешний генератор, все очень гибко конфигурируется! Скорость с которой тикает внутренний генератор зависит от настроек микроконтроллера и обвязки.

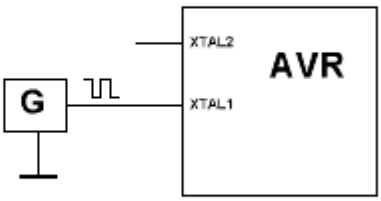


Внешний высокочастотный кварц
1...20Мгц (верхний предел зависит от типа МК)



F=1/(3RC)
C=22...36pF

Внешняя RC цепочка.



Внешний тактовый генератор

Генератор может быть:

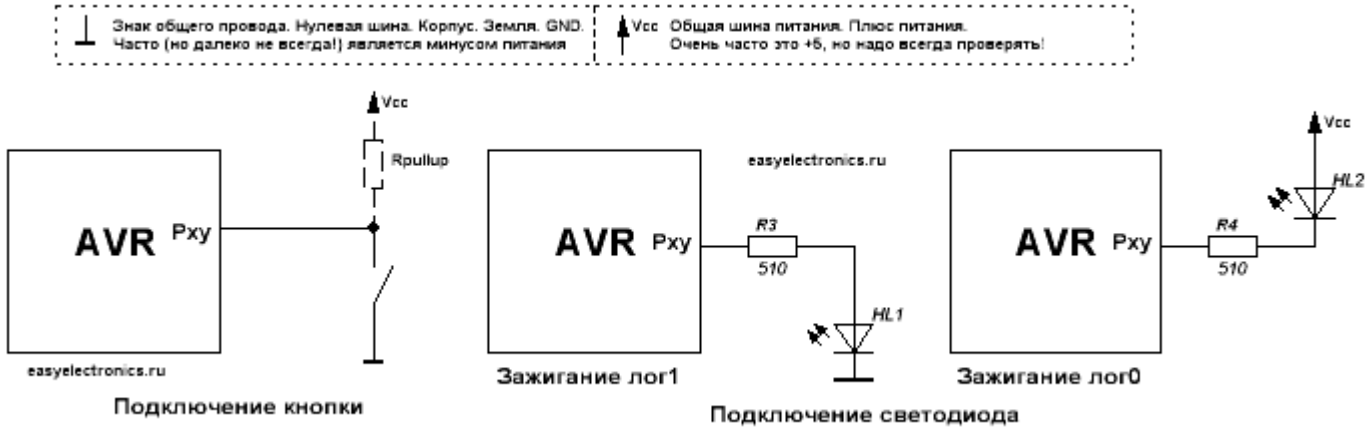
- Внутренним с внутренней задающей RC цепочкой.
  В таком случае никакой обвязки не требуется вообще! А выводы XTAL1 и XTAL2 можно не подключать вовсе, либо использовать их как обычные порты ввода вывода (если МК это позволяет). Обычно можно выбрать одно из 4х значений внутренней частоты. **Этот режим установлен по дефолту**.
- Внутренним с внешней задающей RC цепочкой.
  Тут потребуется подключить снаружи микроконтроллера конденсатор и резистор. Позволяет менять на ходу тактовую частоту, просто подстраивая значение резистора.
- Внутренним с внешним задающим кварцем.
  Снаружи ставится кварцевый резонатор и пара конденсаторов. Если кварц взят низкочастотный (до 1МГц) то конденсаторы не ставят.
- Внешним.
  С какого либо другого устройства идет прямоугольный сигнал на вход МК, который и задает такты. Полезен этот режим, например, если надо чтобы у нас несколько микроконтроллеров работали в жестком синхронизме от одного генератора.

У разных схем есть разные достоинства:
В случае **внутренней RC цепи** мы экономим место на плате, нам не нужно дополнительных деталек, но мы не можем развить максимальную частоту и частота немного зависит от температуры, может плавать.
У внешнего кварца отличные показатели точности, но он стоит лишних 15 рублей и требует дополнительных деталей и, что самое обидное, часто съедает пару ног I/O. Также на внешнем же кварце можно добиться максимальной производительности от МК. Частота МК определяется частотой на которую заточен выбранный кварц. **Внешная RC цепь** позволяет тикать генератору МК быстрей чем от внутренней, стоит дешевле кварца, но имеет те же проблемы со стабильностью частоты, что и внутренняя RC цепь.

Способы тактования МК описаны в даташите в разделе **System Clock and Clock Options** и всецело определяются конфигурацией **Fuse Bit's**. Пока же я настоятельно рекомендую **НЕ ТРОГАТЬ FUSE** пока ты не будешь твердо знать что ты делаешь и зачем. Т.к. выставив что нибудь не то, можно очень быстро превратить МК в кусок бесполезного кремния, вернуть к жизни который будет уже очень непросто (но возможно!)

Подключение к микроконтроллеру светодиода и кнопки
Сам по себе, без взаимодействия с внешним миром, микроконтроллер не интересен — кому интересно что он там внутри себя тикает? А вот если можно как то это отобразить или на это повлиять…

Итак, кнопка и светодиод подключаются следующим образом:



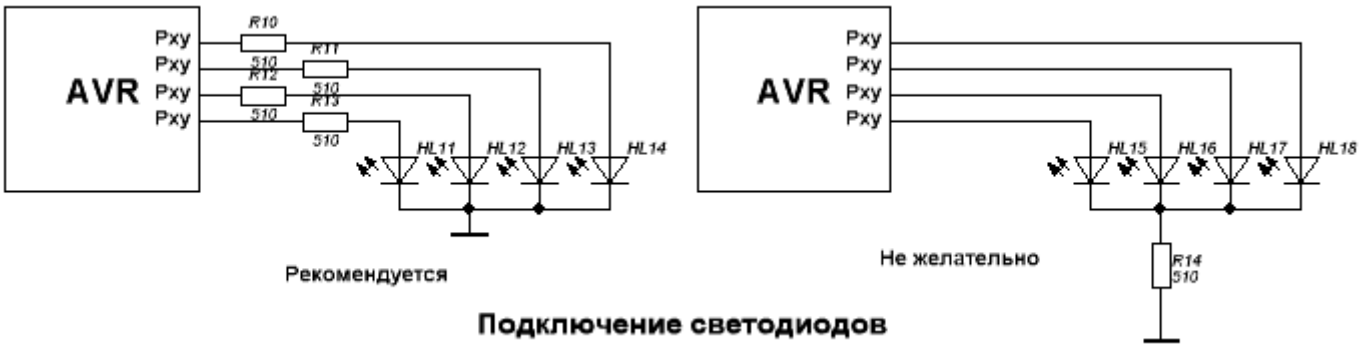
Для кнопки надо выбраную ножку I/O подключить через кнопку на землю. Сам же вывод надо сконфигурировать как **вход с подтяжкой** (DDRxy=0 PORTxy=1). Тогда, когда кнопка не нажата, через подтягивающий резистор, на входе будет высокий уровень напряжения, а из бит **PINxy** будет при чтении отдавать 1. Если кнопку нажать, то вход будет положен на землю, а напряжение на нем упадет до нуля, а значит из **PINxy** будет читаться 0. По нулям в битах регистра **PINx** мы узнаем что кнопки нажаты.
Пунктиром показан дополнительный подтягивающий резистор. Несмотря на то, что внутри AVR на порт можно подключить подтяжку, она слабоватая — 100кОм. А значит ее легко придавить к земле помехой или наводкой, что вызовет ложное срабатывание. А еще эти внутренние подтягивающие резисторы очень любят гореть от наводок. У меня уже с десяток микроконтроллеров с убитыми PullUp резисторами. Все работает, но только нет подтяжки — сгорела. Вешаешь снаружи резистор и работает как ни в чем ни бывало. Поэтому, для ответственных схем я настоятельно рекомендую добавить внешнюю подтяжку на 10кОм — даже если внутреннюю накроет, внешняя послужит. В процессе обучения на это можно забить.

Светодиод подключается на порт двумя способами. По схеме **Порт-земля** или **Порт-Питание**. В первом случае для зажигания диода надо выдать в порт лог1 — высокий уровень (примерно равен Vcc). Во втором случае для зажжения диода требуется выдать в порт лог0 — низкий уровень (около нуля). Для **AVR** разницы вроде бы нет, а вот многие старые серии микроконтроллеров вниз тянули куда лучше чем вверх, так что схема Порт-Питание распространена чаще. Я применяю и ту и другую схему исходя из удобства разводки печатной платы. Ну, а на программном уровне разницы особой нет.
Вывод порта для работы со светодиодом надо сконфигурировать на **выход** (DDRxy=1) и тогда в зависимости от значения в PORTxy на ножке будет либо высокий либо низкий уровень напряжения.

Светодиод **надо подключать через резистор**. Дело в том, что прямое сопротивление светодиода очень мало. И если не ограничивать ток через него, то он просто напросто может сгореть нафиг. Либо, что вероятней, пожечь вывод микроконтроллера, который, к слову, может тянуть что то около 20-30мА. А для нормального свечения обычному светодиоду (всякие термоядерные ультраяркие прожектора [3] мы не рассматриваем сейчас, эти монстры могут и ампер сожрать) надо около 3…15мА.

Так что, на вскидку, считаем:

- Напряжение на выходе ноги МК около 5 вольт, падение напряжени на светодиоде обычно около 2.5 вольт (выше нельзя, иначе диод сожрет тока больше чем надо и подавится, испустив красивый дым)
- Таким образом, напряжение которое должен взять на себя ограничительный резистор будет 5-2.5 = 2.5В.
- Ток нам нужен 5мА — нефига светодиод зря кормить, нам индикация нужна, а не освещение :)
- R=U/I= 2.5/5E-3 = 500Ом. Ближайший по ряду это 510 Ом. Вот его и возьмем. В принципе, можно ставить от 220 Ом до 680 Ом что под руку попадется — гореть будет нормально.



Подключение светодиодов

Если надо подключить много светодиодов, то на каждый мы вешаем по собственному резистору. Конечно, можно пожадничать и поставить на всех один резистор. Но тут будет западло — резистор то один, а диодов много! Соответственно чем больше диодов мы запалим тем меньше тока получит каждый — ток от одного резистора разделится между четырьмя. А поставить резистор поменьше нельзя — т.к. при зажигании одного диода он получит порцию тока на четверых и склеит ласты (либо пожгет порт).

Немного схемотехнических извратов или пара слов о экономии выводов

То что не удается запаять приходится программировать. (С) народная мудрость.

Очень часто бывает так, что вроде бы и памяти контроллера под задачу хватает с лихвой, и быстродействия через край, а ножек не хватает. Вот и приходится ставить избыточный и более дорогой микроконтроллер только потому, что у него банально больше выводов. Покажу парочку примеров как можно за счет усложнения программного кода сэкономить на железе.

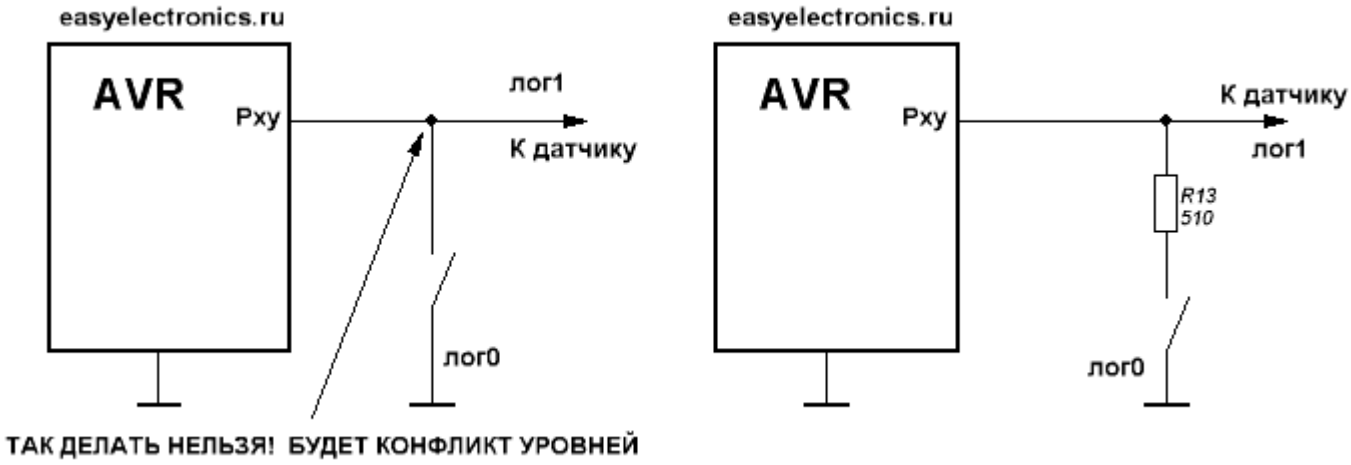
Во главу угла такой экономии обычно ставится принцип динамического разделения назначения выводов во времени. То есть, например, вывод может работать на какую-либо шину, а когда шина не активна, то через этот же вывод можно проверить состояние кнопки, или что нибудь передать по другой шине. Быстро (десятки или даже тысячи раз в секунду) переключаясь между двумя разными назначениями можно добиться эффекта «одновременной работы».

Главное, тут следовать двум правилам:

- Два разных применения не должны мешать друг другу т.е. разделение во времени должно быть построено таким образом, чтобы смежная функция не искажала результат работы проверяемой функции.
- Ни в коем случае нельзя допускать конфликта уровней напряжений.

Приведу пример:

- У есть у нас вывод на который повешан выход с некого датчика и кнопка. Выход с датчика может быть 0, 1 в активном режиме и Hi-Z когда на датчик не приходит сигнал Enable.
- Кнопка же дает на линию жесткий 0, путем короткого замыкания.



Как это должно работать:
Скажем, основную часть времени у нас ввод микроконтроллера настроен на вход Hi-Z и мы снимаем показания с датчика на который подан еще и сигнал Enable. Когда нам надо опросить кнопку, то мы отбираем у датчика Enable и его выходы становятся в режим Hi-Z и нам не мешают. Вывод микроконтроллера мы переводим в режим Pull-Up и проверяем нет ли на входе нуля — сигнал нажатой кнопки. Проверили? Переводим вход МК в Hi-Z вход и подаем Enable на датчик снова. И так много раз в секунду.

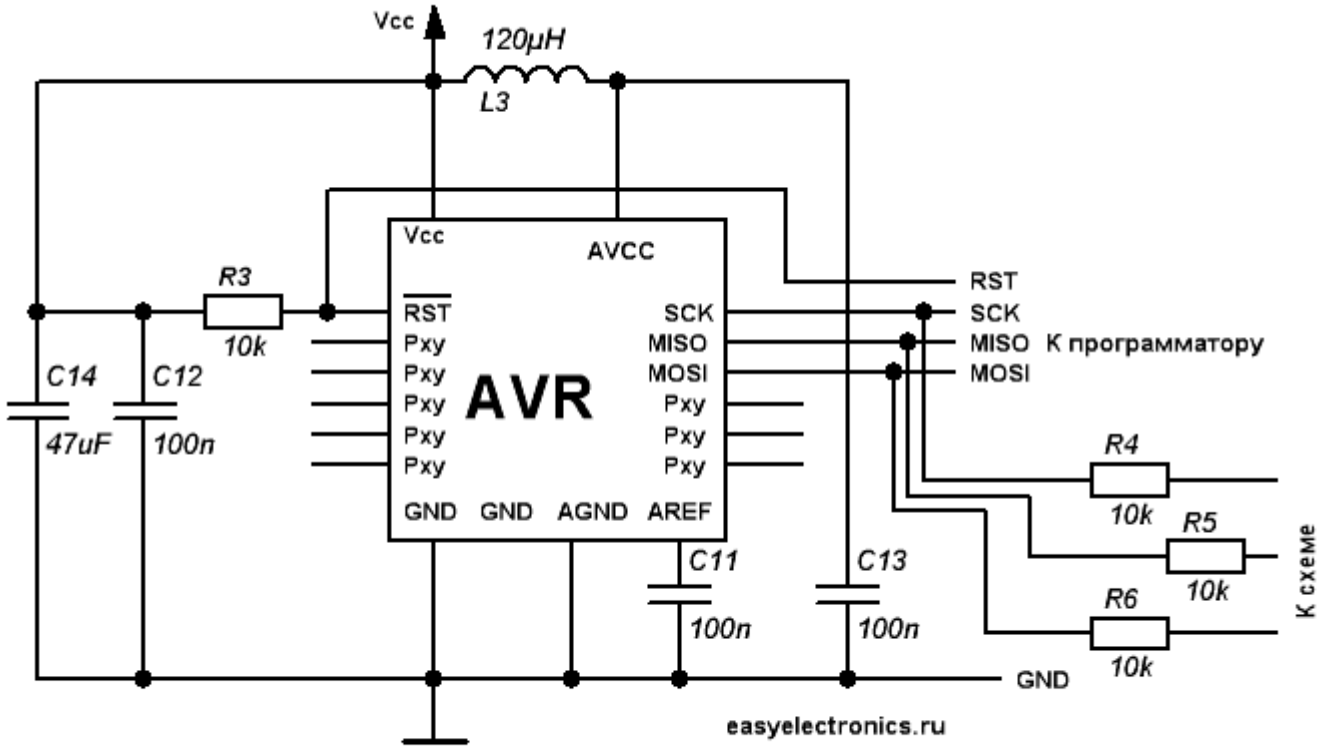
Тут у нас возникает два противоречия:

- **Логическое противоречие**
  0 на линии может быть в двух случаях от датчика или от кнопки. Но в этом случае, пользуясь здравым смыслом и требуемым функционалом, мы логическое противоречие можем не брать во внимание.
  Просто будем знать, что нажатие кнопки искажает показания датчика, а значит когда датчик работает — мы кнопку жать не будем. А чтобы показания датчика не принять за нажатие кнопки мы, в тот момент когда ждем данные с датчика, просто не опрашиваем кнопку. От тупых действий, конечно, это не защитит. Но для упрощения примера защиту от дурака я сейчас во внимания не беру.
- **Электрическое противоречие**
  Если датчик выставит 1, а мы нажмем кнопку, то очевидно, что GND с Vcc в одном проводе не уживутся и кто нибудь умрет. В данном случае умрет выход датчика, как более слабый — куда там хилому транзистору тягаться с медной кнопкой.
  Организационными методами такое противоречие не решить — на глаз нельзя определить напряжение на линии и решить можно жать кнопку или нет. Да и в каком месте сейчас программа можно тоже только догадываться. Поэтому решать будем схемотехнически.
  Добавим резистор в цепь кнопки, резистор небольшой, рассчитывается исходя из максимального тока самого слабого вывода линии.
  Если у нас, например, вывод датчика может дать не более 10мА, то резистор нужен такой, чтобы ток через него от Vcc до GND не превышал этой величины. При питании 5 вольт это будет 510Ом. Теперь, даже если на линии со стороны датчика будет лог1, высокий уровень, то нажатие на кнопку не вызовет даже искажения логического уровня т.к. резистор рассчитан с учетом максимальной нагрузки порта

Пример получился немного сумбурный, но суть думаю понятна. Я хочу чтобы ты увидел и понял не только как делается, но и зачем это делается :)

Ну и несколько примеров нескольких функций на одной ноге:
Во-первых, **ISP разьем**. Я уже давным давно забыл что такое тыкать микроконтроллер вначале в колодку программатора, потом в плату, потом обратно и так по многу раз, пока прогу не отладишь. У меня на плате торчат 6 выводов ISP разьема и при отладке программатор вечно воткнут в плату, а программу я перешиваю порой по нескольку раз в 10 минут. Прошил — проверил. Не работает? Подправил, перепрошил еще раз… И так до тех пор пока не заработает. Ресурс у МК на перепрошивку исчисляется тысячами раз. Но ISP разьем сжирает выводы. Целых 3 штуки — **MOSI, MISO, SCK.**
В принципе, на эти выводы можно еще повесить и кнопки. В таком случае никто никому мешать не будет, главное во время прошивки не жать на эти кнопки. Также можно повесить и светодиоды (правда в этом случае простейший программатор Громова [4] может дать сбой, а вот USBasp [5] молодцом!) тогда при прошивке они будут очень жизнерадостно мерцать :)))

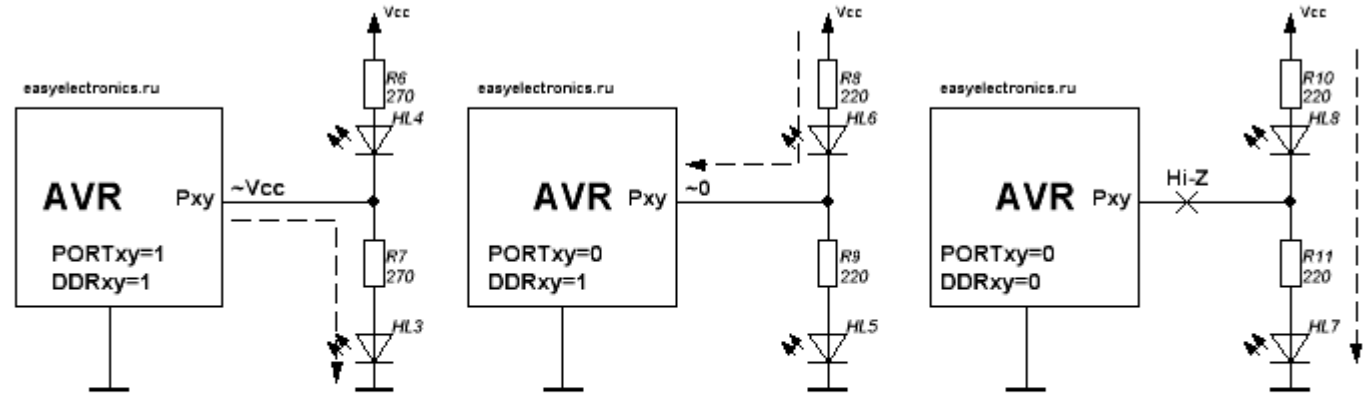
На линии под ISP можно повесить и что нибудь другое, главное, **чтобы при прошивке это ЧТОТО не начало ВНЕЗАПНО чудить**. Например, управление стокилограммовым манипулятором висит на линии ISP и во время прошивки на него пошла куча бредовых данных — так он может свихнуться и кому нибудь бошку разнести. Думать надо, в общем. А вот с каким нибудь LCD вроде HD44780 [6], который работает по шинному интерфейсу прокатит такая схема:



Согласование линий ISP и схемы

Резисторам в 10к отделяем линии программатора от основной схемы. В таком случае, даже если там будут какие либо другие логические уровни, то программатор их легко пересилит и спокойно прошьет микросхему. А при нормальной работе шины эти 10к резисторы особо влиять не будут.

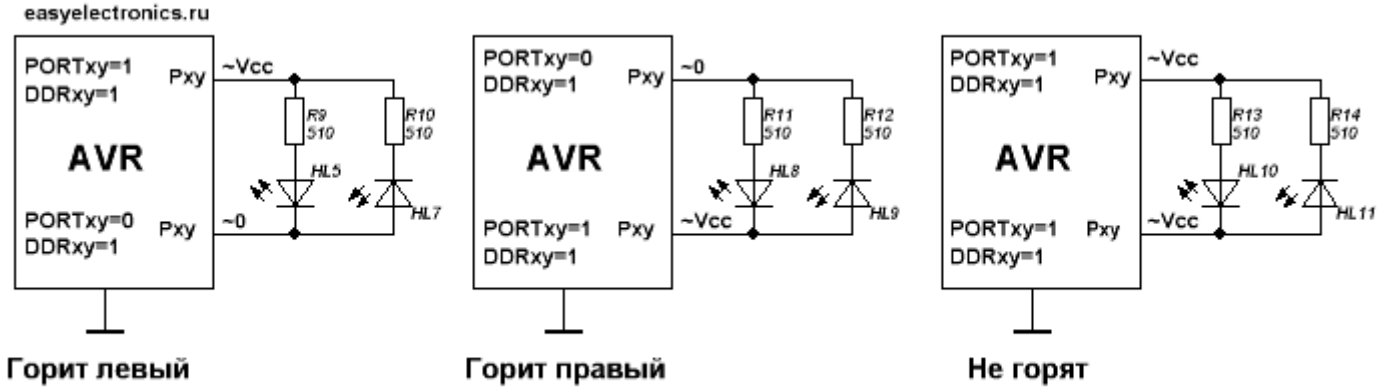
Ножки можно зажать, например, на светодиодах:



Два светодиода на один порт

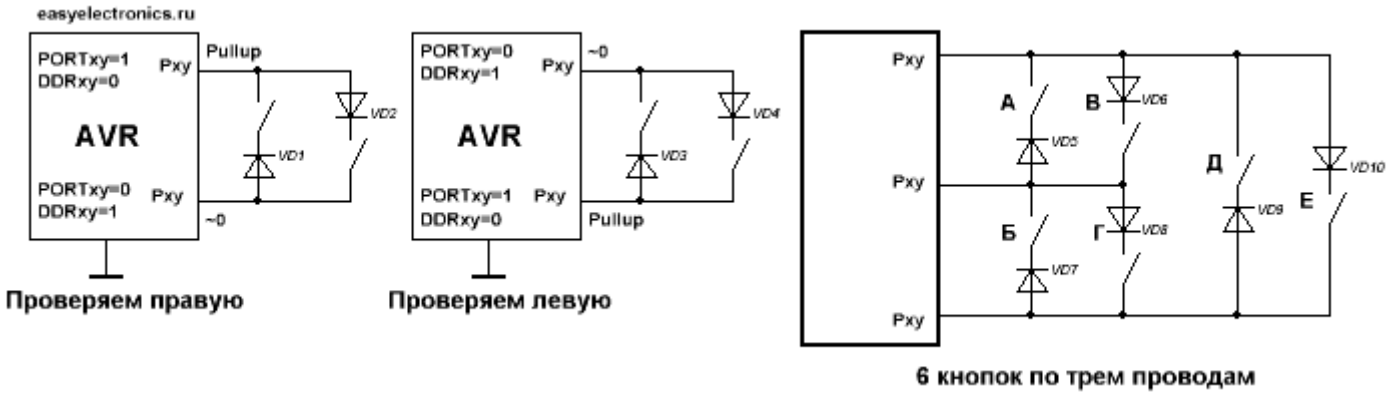
Переключаем выход с 0 на 1 и зажигаем то верхний то нижний диод. Если надо зажечь оба, то мы просто переводим вывод микроконтроллера в режим **Hi-Z** и словно нет его, а диоды будут гореть сквозным током. Либо быстро быстро переключать диоды между собой, в этом случае на глаз они будут оба гореть. Недостаток схемы очевиден — диоды нельзя погасить. Но если по задумке хотя бы один должен гореть, то почему бы и нет? **UPD:** Тут подумал, а ведь можно подобрать светодиоды и резисторы так, чтобы их суммарное падение напряжения было на уровне напряжения питания, а суммарные резисторы в таком случае загонят ток в такой мизер, что когда нога в Hi-Z то диоды вообще гореть не будут. По крайней мере на глаз это будет не заметно совсем. Разве что в кромешной тьме.

Следующий вариант он не дает экономию ножек, зато позволяет упростить разводку печатной платы, не таща к двум диодам еще и шину питания или земли:



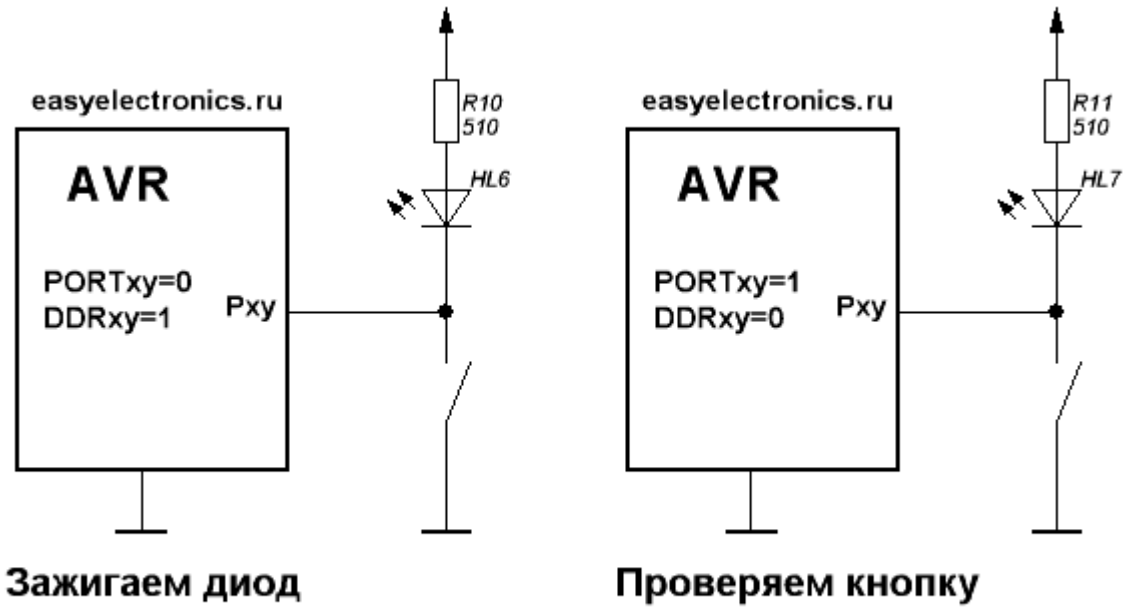
Подключение светодиодов по схеме ПОРТ-ПОРТ

Тут все просто — превращая один из выводов то в 0 то в 1 гоняем ток то в одну сторону то в другую. В результате горит то один то другой диод. Для погашения обоих — переводим ноги в какое то единое положение 11 или 00. Два диода сразу зажечь не получится, но можно сделать динамическую индикацию — если их быстро быстро переключать, то глаз не заметит подставы, для него они будут оба горящими. А добавив третью линию можно по трем ногам прогнать до 6 светодиодов на том же принципе.



А применив сходную тактику к кнопкам можно либо упростить разводку, либо по трем ножкам развести 6 кнопок. Тут тоже все просто — одна нога дает подтяг, вторая косит под землю. Нажатие кнопки дает просадку напряжения на подтягивающей ножке. Это чует программа, поочередно опрашивающая каждую кнопку. Потом роли ножек меняются и опрашивается следующая кнопка.
В шестикнопочном режиме ситуация схожая — одна ножка дает подтяг, другая землю, а третья прикидывается ~~ветошью~~ Hi-Z и не отсвечивает. Но тут есть один побочный эффект. Например, опрашиваем мы кнопку «В». Для этого у нас верхняя линия встает на **вход с подтяжкой** (PORTxy=1, DDRxy=0), средня дает **низкий уровень на выходе** (PORTxy=0, DDRxy=1), нижняя не участвует в процессе ибо стоит в **Hi-Z** (PORTxy=0, DDRxy=0). Если мы нажмем кнопку «В» то верхняя линия в этот момент просядет и программа поймет что нажата кнопка «В», но если мы не будем жать «В», а нажмем одновременно «Е» и «Б» то верхняя линия также просядет, а программа подумает что нажата «В», хотя она там и рядом не валялась. Минусы такой схемы — возможна неправильная обработка нажатий. Так что если девайсом будут пользоваться быдло-операторы, жмущие на все подряд без разбора, то от такой схемы лучше отказаться.

Ну и, напоследок, схема показывающая как можно обьединить кнопку и светодиод:



Обьединение кнопки и светодиода

Работает тоже исключительно в динамике. То есть все время мы отображаем состояние светодиода — то есть выдаем в порт либо 0 (диод горит) либо Hi-Z (диод не горит). А когда надо опросить кнопку, то мы временно (на считанные микросекунды) переводим вывод в режим вход с подтягом (DDRxy=0 PORTxy=1) и слушаем кнопку. Режим когда на выводе сильный высокий уровень (DDRxy=1 PORTxy=1) включать ни в коем случае нельзя, т.к. при нажатии на кнопку можно пожечь порт.

Минусы — при нажатии на кнопку зажигается светодиод как ни крути. Впрочем, это может быть не багой, а фичей :)

Вот такие пироги. А теперь представьте себе прогу в которой реализованы все эти динамические фичи + куча своего алгоритма. Выходит либо бесконечная череда опросов, либо легион всяких флагов. В таких случаях простейшая диспетчеризация или кооперативная RTOS [7] это то что доктор прописал — каждый опрос гонишь по циклу своей задачи и не паришься. Зато юзаешь везде какую-нибудь ATTiny2313 и ехидно глядишь на тех кто в ту же задачу пихает Mega8 или что пожирней :)

Я ничего не знаю и боюсь что либо сжечь, что мне делать???
Не бояться и делать. В конце концов, микроконтроллер не такая уж дорогая вещь чтобы сокрушаться по поводу его смерти. Выкинул в помойку и достал из пакетика новый. На худой конец, если совсем уж страшно, то можно купить готовую демоплату на которой все уже спаяно и разведено как надо. Тебе останется только программировать и смотреть результат.
А потом, на примере того как сделана демоплата, попробовать сделать что то свое. Сама же демоплата представляет собой микроконтроллер + немного стартовой периферии, которой хватит на ряд несложных опытов и которая может облегчить подключение и исследование других устройств. Демоплаты есть разные, например фирменные комплексы вроде STK500 или AVR Butterfly или моя **Pinboard** [8] которая была спроектированна исходя из моего опыта и на которой будет строится весь дальнейший учебный курс.

Ссылки по теме:
Как управлять через микроконтроллер электромагнитным реле. [9]
Как подключить к микроконтроллеру что либо гораздо более мощное чем светодиод. [10]
Как управлять через микроконтроллер мощной нагрузкой переменного тока. [11]
Как подключить к микроконтроллеру текстовый ЖК дисплей на контроллере HD44780. [6]
Как подключить к микроконтроллеру ОЧЕНЬ МНОГО КНОПОК — Матричная клавиатура [12]
Как замерить микроконтроллером аналоговый сигнал (Использование АЦП) [13]
Как связать микроконтроллер и компьютер (по проводам, радио каналу, локальной сети) [14]
Как научить микроконтроллер отличать свет от тьмы — трактат о фотодатчиках [15]

З.Ы.
Камрад Dsiss снял видео о том, как Мега48 без подтяжки RESET сбрасывается от касания пальцем:

Подтягивайте RESET!!!

AVR. Учебный курс. Трактат о программаторах

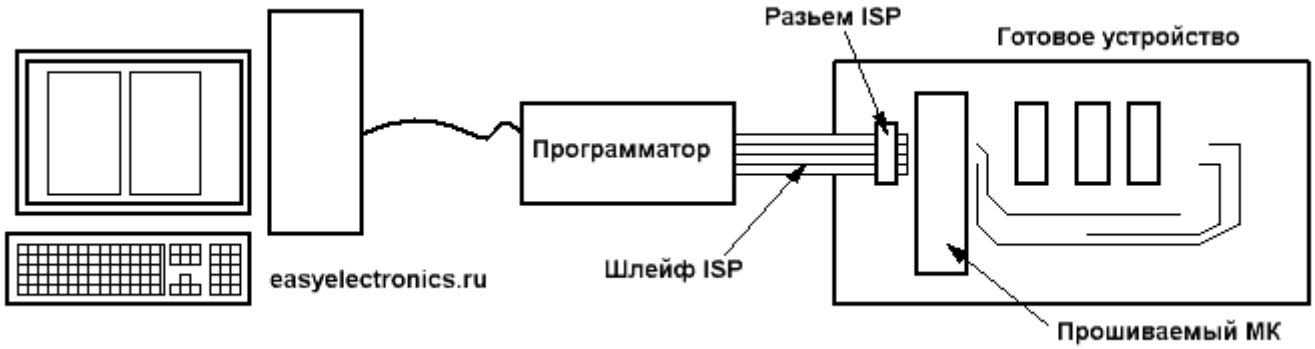
Программа для микроконтроллера пишется на любом удобном языке программирования, компилируется в бинарный файл (или файл формата intel HEX) и заливается в микроконтроллер посредством программатора.

Итак, первым шагом в освоении микроконтроллера обычно становится программатор. Ведь без программатора невозможно загнать программу в микроконтроллер и он так и останется безжизненным куском кремния.

Что же представляет из себя это устройство?
В простейшем случае программатор это девайс который связывает микроконтроллер и компьютер, позволяя с компа залить файл прошивки в память контроллера. Также нужна прошивающая программа, которая по специальному протоколу загонит данные в микроконтроллер.

Программаторы бывают разные под разные семейства контроллеров существуют свои программаторы. Впрочем, бывают и универсальные. Более того, даже ту же простейшую AVR'ку можно прошить несколькими способами:

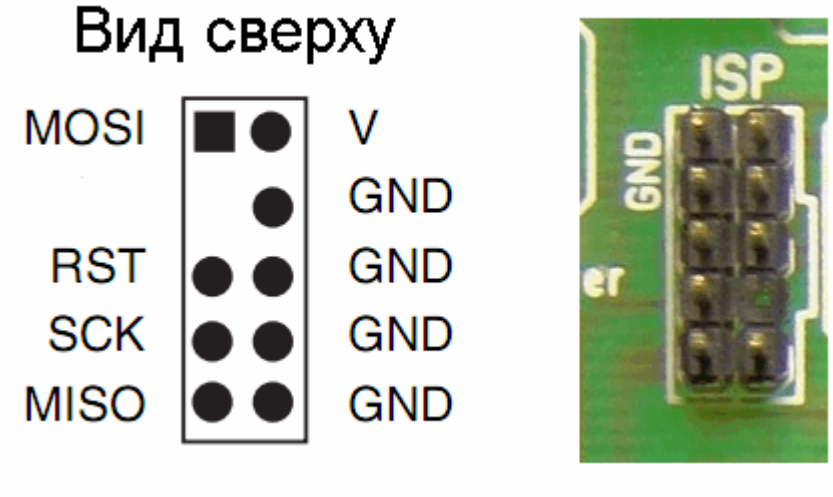
Внутрисхемное программирование (ISP)
Самый популярный способ прошивать современные контроллеры. Внутрисхемным данный метод называется потому, что микроконтроллер в этот момент находится в схеме целевого устройства — он может быть даже наглухо туда впаян. Для нужд программатора в этом случае выделяется несколько выводов контроллера (обычно 3..5 в зависимости от контроллера).



К этим выводам подключается прошивающий шнур программатора и происходит заливка прошивки. После чего шнур отключается и контроллер начинает работу.
У AVR прошивка заливается по интерфейсу SPI и для работы программатора нужно четыре линии и питание (достаточно только земли, чтобы уравнять потенциалы земель программатора и устройства):

- MISO — данные идущие от контроллера (Master-Input/Slave-Output)
- MOSI — данные идущие в контроллер (Master-Output/Slave-Input)
- SCK — тактовые импульсы интерфейса SPI
- RESET — сигналом на RESET программатор вводит контроллер в режим программирования
- GND — земля

Сам же разъем внутрисхемного программирования представляет собой всего лишь несколько штырьков. Лишь бы на него было удобно надеть разъем. Конфигурация его может быть любой, как тебе удобней.
Однако все же есть один популярный стандарт:



Для внутрисхемной прошивки контроллеров AVR существует не один десяток разнообразных программаторов. Отличаются они в первую очередь по скорости работы и типу подключения к компьютеру (COM/LPT/USB). А также бывают безмозглыми или со своим управляющим контроллером.

Безмозглые программаторы, как правило, дешевые, очень простые в изготовлении и наладке. Но при этом обычно работают исключительно через архаичные COM или LPT порты. Которые найти в современном компьютере целая проблема. А еще требуют прямого доступа к портам, что уже в Windows XP может быть проблемой. Плюс бывает зависимость от тактовой частоты процессора компьютера.

Так что твой 3ГГЦ-овый десятиядерный монстр может пролететь, как фанера над Парижем.

Идеальный компьютер для работы с такими программаторами это какой-нибудь PIII-800Mhz с Windows98…XP.
Вот очень краткая подборка проверенных лично безмозглых программаторов:

- **Программатор Громова** [1].
  Простейшая схема, работает через оболочку UniProf(удобнейшая вещь!!!), но имеет ряд проблем. В частности тут COM порт используется нетрадиционно и на некоторых материнках может не заработать. А еще на быстрых компах часто не работает. Да, через адаптер USB-COM эта схема работать не будет. По причине извратности подхода :)
- **STK200** [2]
  Надежная и дубовая, как кувалда, схема. Работает через LPT порт. Поддерживается многими программами, например avrdude. Требует прямого доступа к порту со стороны операционной системы и наличие LPT порта.
- **FTBB-PROG** [3].
  Очень надежный и быстрый программатор работающий через USB, причем безо всяких извратов. С драйверами под разные операционные системы. И мощной оболочкой avrdude. Недостаток один — содержит редкую и дорогую микросхему FTDI FT232RL, да в таком мелком корпусе, что запаять ее без меткого глаза, твердой руки и большого опыта пайки весьма сложно. Шаг выводов около 0.3мм. Данный программатор встроен в демоплату Pinboard [4]

Программаторы с управляющим контроллером лишены многих проблем безмозглых. Они без особых проблем работают через USB. А если собраны на COM порт, то без извращенских методик работы с данными — как честный COM порт. Так что адаптеры COM-USB работают на ура. И детали подобрать можно покрупней, чтобы легче было паять. Но у этих программаторов есть другая проблема — для того чтобы сделать такой программатор нужен другой программатор, чтобы прошить ему управляющий контроллер. Проблема курицы и яйца. Широко получили распространение такие программаторы как:

- **USBASP** [5]
- AVRDOPER
- AVR910 Protoss

Внутрисхемное программирование, несмотря на все его удобства, имеет ряд ограничений.
Микроконтроллер должен быть запущен, иначе он не сможет ответить на сигнал программатора. Поэтому если неправильно выставить биты конфигурации (FUSE), например, переключить на внешний кварцевый резонатор, а сам кварц не поставить. То контроллер не сможет запуститься и прошить его внутрисхемно будет уже нельзя. По крайней мере до тех пор пока МК не будет запущен.
Также в битах конфигурации можно отключить режим внутрисхемной прошивки или превратить вывод RESET в обычный порт ввода-вывода (это справедливо для малых МК, у которых RESET совмещен с портом). Такое действо тоже обрубает программирование по ISP.

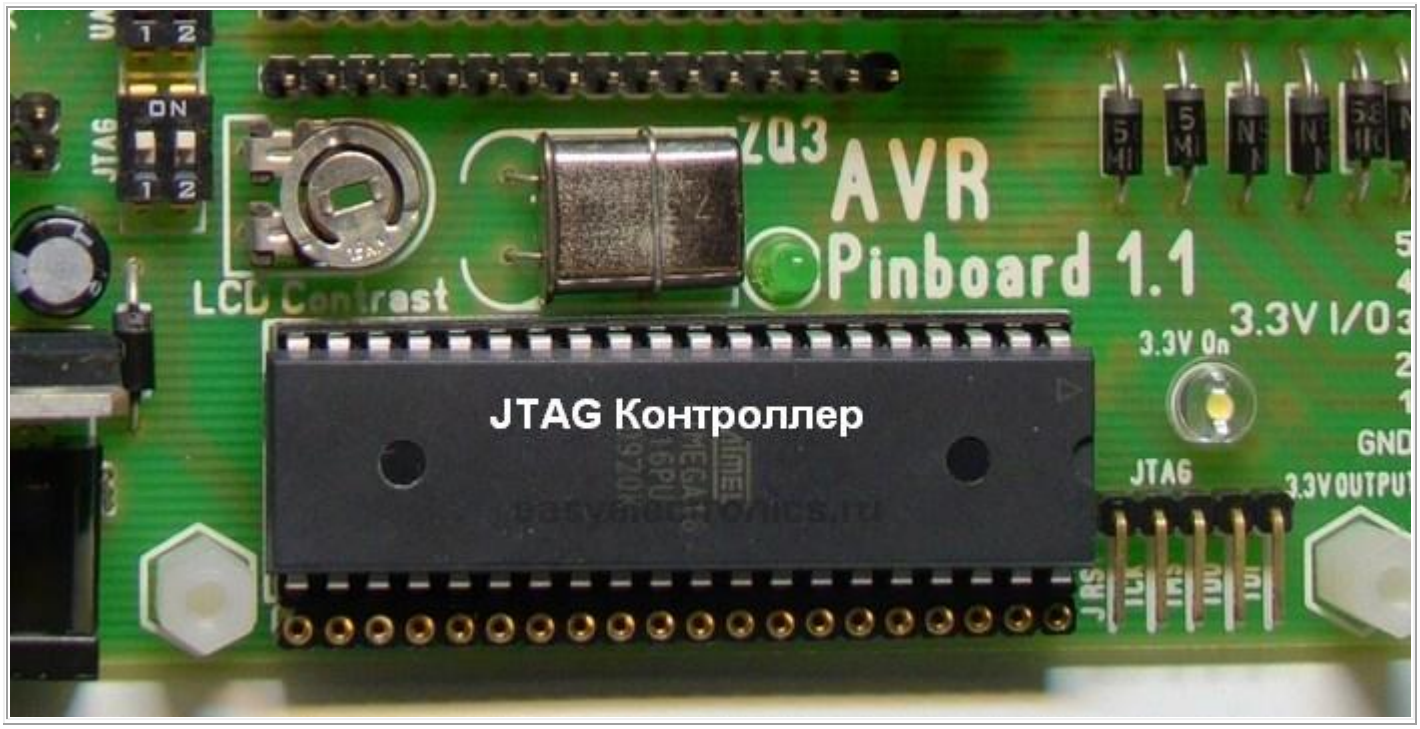
Параллельное высоковольтное программирование
Обычно применяется на поточном производстве при массовой (сотни штук) прошивке чипов в программаторе перед запайкой их в устройство.
Параллельное программирование во много раз быстрей последовательного (ISP), но требует подачи на RESET напряжения в 12 вольт. А также для параллельной зашивки требуется уже не 3 линии данных, а восемь + линии управления. Для программирования в этом режиме микроконтроллер вставляется в панельку программатора, а после прошивки переставляется в целевое устройство.

Для радиолюбительской практики он особо не нужен, т.к. ISP программатор решает 99% насущных задач, но тем не менее параллельный программатор может пригодиться. Например, если в результате ошибочных действий были неправильно выставлены FUSE биты и был отрублен режим ISP. Параллельному программатору на настройку FUSE плевать с высокой колокольни. Плюс некоторые старые модели микроконтроллеров могут прошиваться только высоковольтным программатором.
Из параллельных программаторов для AVR на ум приходит только:

- HVProg от ElmChan
- Paraprog
- DerHammer

А также есть универсальные вроде TurboProg 6, BeeProg, ChipProg++, Fiton которые могут прошивать огромное количество разных микроконтроллеров, но и стоят неслабо. Тысяч по 10-15. Нужны в основном только ремонтникам, т.к. когда не знаешь что тебе завтра притащат на ремонт надо быть готовым ко всему.

Прошивка через JTAG
Вообще **JTAG** это отладочный интерфейс [6]. Он позволяет пошагово выполнять твою программу прям в кристалле. Но с его помощью можно и программу прошить, или FUSE биты вставить. К сожалению JTAG доступен далеко не во всех микроконтроллерах, только в старших моделях в 40ногих микроконтроллерах. Начиная с Atmega16.
Компания AVR продает фирменный комплект JTAG ICEII для работы с микроконтроллерами по JTAG, но стоит он (как и любой профессиональный инструмент) недешево. Около 10-15тыр. Также есть первая модель JTAG ICE. Ее можно легко изготовить самому, а еще она встроена в мою демоплату **Pinboard** [4]



Прошивка через Bootloader
Многие микроконтроллеры AVR имеют режим самопрошивки. Т.е. в микроконтроллер изначально, любым указанным выше способом, зашивается спец программка — bootloader. Дальше для перешивки программатор не нужен. Достаточно выполнить сброс микроконтроллера и подать ему специальный сигнал. После чего он входит в режим программирования и через обычный последовательный интерфейс в него заливается прошивка. Подробней описано в статье посвященной бутлоадеру [7].
Достоинство этого метода еще и в том, что работая через бутлоадер очень сложно закосячить микроконтроллер настолько, что он не будет отвечать вообще. Т.к. настройки FUSE для бутлоадера недоступны.
Бутлоадер также прошит по умолчанию в главный контроллер демоплаты **Pinboard** [4] чтобы облегчить и обезопасить первые шаги на пути освоения микроконтроллеров.

SinaProg — графическая оболочка для AVRDude

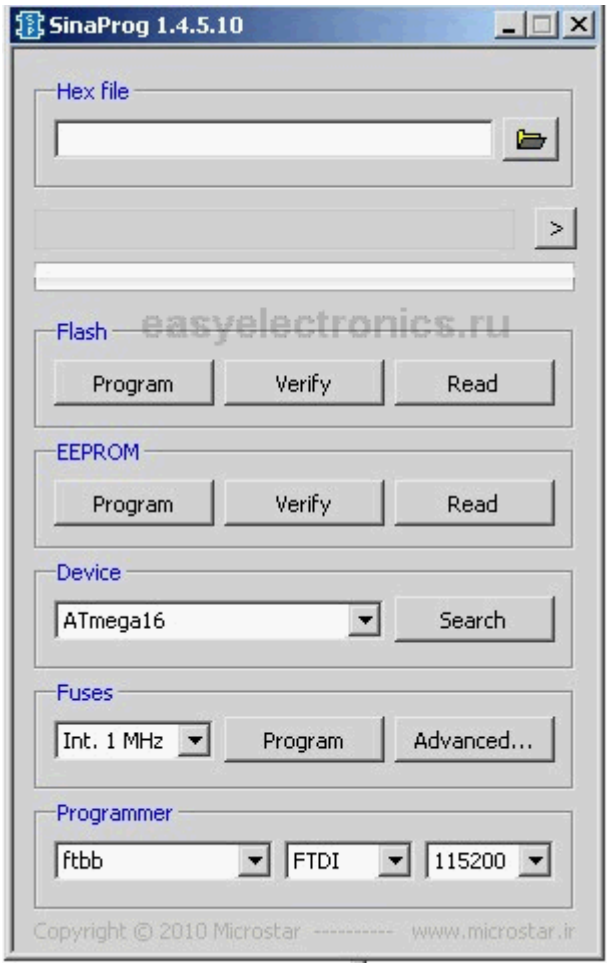
Вот уже много лет я пользуюсь мощнейшей программой для прошивки — **avrdude**. Программа эта поддерживает почти все виды программаторов, а те что не поддерживает изначально легко в нее добавляются.
Но есть у ней особенность которая сильно отпугивает многих — она консольная. И все шаманства с ней заключаются в формировании командной строки.

С одной стороны это хорошо — консольная утилита позволяет очень сильно ускорить и автоматизировать процесс прошивки. Один раз написал батничек и для перешивки только вызывать его и все.

Остается проблема прошивки разных устройств, ведь под каждую придется делать свой батник. И ладно бы ключи прописать, да файл с прошивкой указать. Самая засада начинается с fuse битами (впрочем, как и везде в мире AVR ;)) Тут их надо вначале внимательно проштудировать в даташите, выписать в виде байтов, вписать правильно в командную строку и молиться на то, чтобы нигде не ошибиться.
Да, для avrdude написано много оболочек, но все что мне встречались раньше решали лишь малозначительную проблему выбора нужного ключа файла и/или программатора, а фузы также было надо указывать числом.

Проблема решилась с появлением новой версии оболочки SinaProg от команды ~~программистов-террористов из Аль-каиды~~ иранских AVR программеров.

Запускаем… Если не запустилась, то возможно потребуется установить фреймворк от NI — LabView RunTime Library [1]



Морда выглядит простенько и со вкусом. Ничего лишнего. Выбираем в первой строке хекс файла и зашиваем его в нужную память — flash или eeprom.

Следом идет прогресс бар и кнопка открытия консольного лога — ошибки смотреть.

Ниже выбираем тип микроконтроллера, также есть кнопочка поиска — полезно для проверки работы программатора.

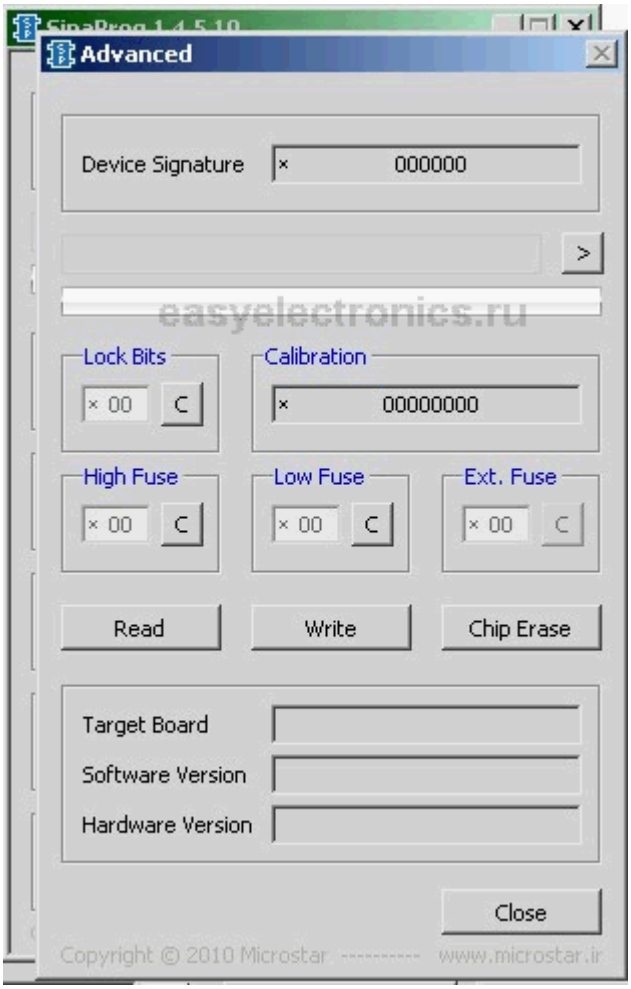
Отдельно стоит сказать про секцию Fuses.
Осторожней с выпадающим списком. С виду там все просто, но это на самом деле предустановки, описываются они файле **Fuse.txt**вот его дефолтное содержание:

```
1  Default.
2  []
3
4  ATmega8.ATmega16.
5  [Int. 1 MHz     d9e1
6  Int. 2 MHz      d9e2
7  Int. 4 MHz      d9e3
8  Int. 8 MHz      d9e4
9  Ext. Crys.      d9ff]
10
11 ATmega32.
12 [Int. 1 MHz     d9e1
13 Int. 2 MHz      d9e2
14 Int. 4 MHz      d9e3
15 Int. 8 MHz      d9e4
16 Ext. Crys.      d9ff]
```

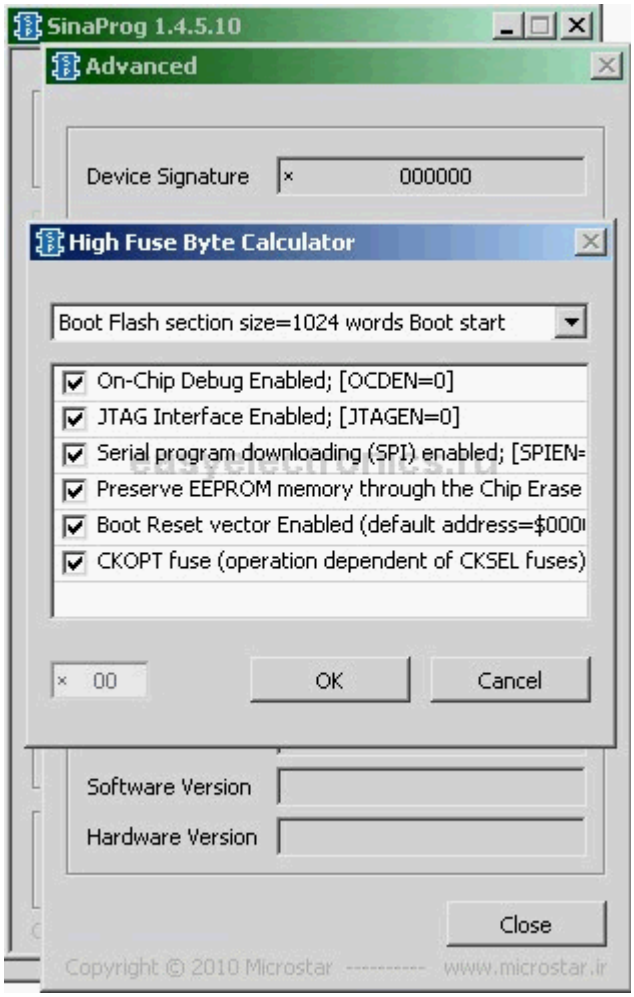
Видишь, формат очень прост. Строка контроллера (обязательно с точкой!) и в квадратных скобаках возможные варианты (отделенные табуляцией) с байтами тех самых фузов. Обратите внимание, что тут меняется **СРАЗУ ОБА БАЙТА** Fuse битов. Т.е. касаются далеко не только тактовой частоты. А еще всего остального что конфигурируется в FUSE. Так что я бы сразу переназвал их иначе. Скажем как
All Default,but 1MHZ
All Default,but 2MHZ
All Default,but 4MHZ

Чтобы было понятней. Но это как бы быстрые шаблоны и не претендуют на глобальность.

Лучше сразу нажать **Advanced** и узреть … тот самый цифровой ввод.



Но не стоит напрягаться, достаточно нажать кнопочку «C» (видимо авторы имели ввиду Calculator) и увидеть удобнейшие выпадающие списки с человеческим описанием. Прям как в STK500.



Но ни в коем случае не забывайте нажать кнопочку READ перед тем как что либо менять. Помните, неустановленные fuse это тоже какое то значение, которое будет записано при нажатии кнопки WRITE

Конфигурирование программы
Впрочем, все же главным достоинством этой оболочки является ее легкая заточка под любой программатор поддерживаемый через **AVRDUDE**.

Покажу на примере программатора встроенного в **Pinboard** [2] (На базе FTDI, но распиновка выводов немного отличная от FTBB описанного ранее [3]).

Закрываем программу, чтобы не мешалась и не держала файлы.

Прописывание конфигурации FTBB в файле avrdude.conf я описывать не буду, подробней я об этом уже писал ранее [3].

Считаем, что это уже сделано и в файле конфигов **avrdude** у нас есть нужная секция. Я назвал ее **pinb**, впрочем, название может быть и произвольным.

Первым делом открываем файл **Programmer.txt** и видим там список:

```
1  ABCmini
2  ALF
3  Arduino
4  AT ISP
5  AVR109
6  AVR910
7  AVR911
8  AVRISP
```

```
9  AVRISP 2
10 AVRISP mkII
11 AVRISP v2
```

и еще полторы страницы все известных и не очень типов программаторов. Вписываем там первой строкой наш pinb

```
1  ABCmini
2  pinb
3  ALF
4  Arduino
5  AT ISP
6  AVR109
7  AVR910
8  AVR911
9  AVRISP
10 AVRISP 2
11 AVRISP mkII
12 AVRISP v2
```

Все, теперь он в списке, но ему еще надо сопоставить порт. Поэтому открывай файл **Port.txt**
Вот его дефолтное содержимое:

```
1  Default.
2  [COM1   com1
3  COM2    com2
4  COM3    com3
5  COM4    com4
6  COM5    com5
7  COM6    com6
8  COM7    com7
9  COM8    com8
10 COM9    com9
11 LPT1    lpt1
12 LPT2    lpt2
13 LPT3    lpt3]
14
15 STK500 v2.
16 [USB    avrdoper]
17
18 AVRISP mkII.
19 [USB    usb]
20
21 USBasp.
22 [USB    x]
```

Как видишь, формат тут сходный. Название программатора (с точкой в конце!), а в скобках варианты. Причем первым делом пишем произвольное название порта, а потом то в каком виде он должен подставиться в командную строку avrdude. Между ними табуляция.

Порт **FTDI bitbang** в консоли называется **ft#** и номер от нуля до бесконечности. В зависимости от того сколько чипов FTDI навешано на твой компьютер в данный момент. Причем учитываются именно подключенные, активные, чипы. И не стоит путать этот номер с номером виртуального COM порта который этот чип организует. Так что если у тебя в системе всего один адаптер USB-COM на базе FTDI, то какой бы там COM порт ни был, для bitbang программатора он зовется ft0 и никак иначе. На всякий случай добавляем несколько вариантов.

Добавляем туда нашу секцию

```
1 pinb.
2 [FTDI   ft0
3 FTDI1   ft1
```

```
4 FTDI2   ft2
5 FTDI3   ft3]
```

Осталась еще одна маленькая деталь. Опция скорости. Для обычных программаторов вполне подойдет и дефолтная настройка, но вот незадача — там используется ключ -b, а он меняет только битрейт COM порта. А если нам надо менять битклок FTDI битбанг эмуляции порта? Тут в avrdude за это отвечает ключ -B, но Sina его не знает. Придется добавлять. За скорость отвечает файл speed.txt

Вписываем туда нашу скорость в нагрузку к уже имеющейся в таком виде:

```
1  1200    x -B 1200
2  2400    x -B 2400
3  4800    x -B 4800
4  9600    x -B 9600
5  14400   x -B 14400
6  19200   x -B 19200
7  38400   x -B 38400
8  57600   x -B 57600
9  115200  x -B 115200
10 230400  x -B 230400
11 460800  x -B 460800
12 921600  x -B 921600
13 3000000 x -B 3000000
```

Я же, поскольку последнее время пользуюсь только FTDI BB Программатором, удалил вообще из этого файла все и оставил только то, что привел выше.

Отлично, программатор мы прописали и порт мы сопоставили. Даже скорости выправили. Теперь надо сину заставить это дело все прожевать. Для этого берем и удаляем файл SinaProg.sav Не знаю как построена логика программы, но почему то именно это действие заставляет ее перечитать собственные конфиги и добавить наши строки в списки.

Все! Готово — можно шить! Удачной прошивки!

Сайт разработчиков SinaProg (увы сдох :( ) [4]
Моя сборка SinaProg [5] с уже настроенными конфигами под **Pinboard** [2] и FTBB

AVR. Учебный Курс. Использование Bootloader'а

Почти все микроконтроллеры серии Mega с памятью от 8КБ могут прошиваться через **бутлоадер**. Фишка удобная и применяется довольно часто, однако подробного мануала как работать с бутлоадером на **AVR** я в свое время не нашел и пришлось разбираться самостоятельно. С той поры мало что изменилось. Пора бы дать подробное описание как выбрать, скомпилировать, прошить и в дальнейшем использовать **bootloader** для AVR.

Ликбез
Что же это такое бут и с чем его едят. **BootLoader** это всего лишь небольшая программка которая сидит в специальной области памяти микроконтроллера и слушает какой-либо интерфейс. Обычно это **UART**, но бывает и **SPI**, **USB** и даже **SoftUSB** бутлоадеры.

При загрузке контроллера управление первым делом передается бутлоадеру и он проверяет есть ли условие для запуска. Условие может быть любым, но обычно это либо наличие спец байта по интерфейсу, либо наличие нужного логического уровня на выбранной ножке контроллера, сигнализирующее о том, что мы хотим обратиться к буту прошивку. Если условие есть — то бутлоадер может, например, принять прошивку по **UART**'у и сам прошить ее во флеш. Или, наоборот, считать прошивку из флеша и выдать через **UART**, считать или записать **EEPROM**, подрыгать ножками. Да что угодно. Но обычно все же с помощью бута осуществляют прошивку микроконтроллера без применения спец программатора.

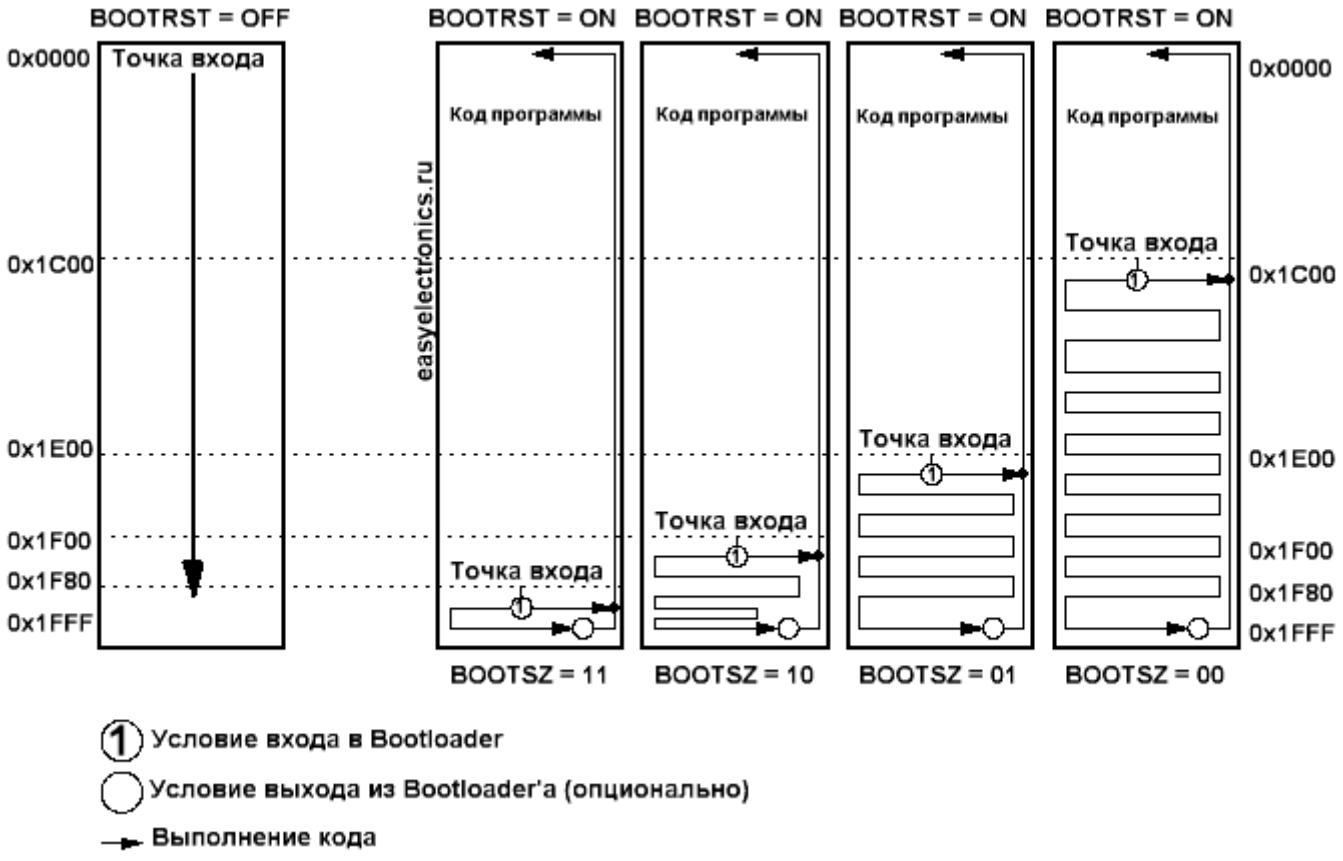
Если разрешающего условия при старте нет, то бут завершает свою работу и передает управление основной программе.

Зачем он нужен вообще?
В самом деле, зачем эти сложности? Зачем тратить и без того малое количество памяти на какую то вспомогательную утилиту? А порой иначе и нельзя! Вот, представь, стоит у тебя девайс где нибудь под землей на большой глубине. Или на высоченной башне куда лезть и лезть, или девайсов у тебя таких миллион? А связь у тебя с девайсом по **UART** какому нибудь или радиоканалу. И вот надо прошить девайс свежей версией прошивки. Выкапывать, лезть на башню или тыкать в каждый из девайсов шнур программатора… это же сдохнуть можно! А так дал девайсу общий сброс, приказал удаленно бутлоадеру всосать новую прошивку и вуаля!
Мало того, с помощью бута можно сильно облегчить обслуживание коммерческих устройств. Например, такая простая вещь как обновление прошивки, которую поддерживает масса девайсов вроде плееров или сотовых телефонов и которую делает обычный юзер. Выходит прошивка доступна и конкурентам? А вот фигу! Она шифрованая, а бутлоадер содержит ключ для ее дешифровки, дешифрует на лету и заливает во флеш. Красота, правда?

Недостатков у любого бутлоадера два — самый главный в том, что он отжирает часть флеша. Это неизбежное зло. Второй, не менее весомый, то что он стартует первый и если не позаботиться о грамотном алгоритме входа в бут, то девайс может сделать это самопроизвольно, записав в себя черт знает что. Такая беда была, ЕМНИП, со старыми GSM модулями **SIM300DZ** у которых часто ВНЕЗАПНО при неправильном выключении питания слетала прошивка. А просто модуль входил в бут режим и наведенные помехи на входах устраивали ему армагедец.

Откуда Bootloader берется в контроллере?
Многие современные МК уже изначально, с завода, идут с прошитым bootloader’ом. В AVR же ничего подобного нет, поэтому его надо прошить самостоятельно. Да, программатор потребуется. Обычный ISP, любой — Громова, STK200, Пять проводков, USBAsp да тысячи их. Им зашиваешь в память обычную прошивку, но содержащую бут. После чего программатор тебе уже не потребуется — дальше с контроллером можно общаться через загрузчик.

Bootloader в AVR
Что происходит при старте контроллера? В нормальном состоянии процессор начинает по одной выполнять инструкции из памяти программ, начиная с нулевого адреса. Вот так это примерно выглядит в ATmega16, в других все аналогично, только адреса другие.



Но если активировать Fuse бит **BOOTRST** то процессор будет стартовать не с нулевого адреса, а с адреса начала Boot сектора. Этот сектор расположен в самом конце памяти программ и его размер задается **FUSE** битами **BOOTSZx**.

Вот как, например, выглядит таблица соответствия битов BOOTSZx и размера загрузочной области для ATMega16:

Обрати внимание на то, что адрес задается в словах. Учитывая то, что одна команда в AVR занимает одно слово, то в самом скромном варианте бутлоадера нам надо уложиться в жалкие 128 команд, а в самом жирном в нашем распоряжении аж 1024 команды. Кроме всего прочего, у бутлоадера есть еще куча разных битов конфигурации разрешающие запись или чтение из памяти программ, организующие защиту от копирования и прочие фишечки. Я их описывать не буду — тут нужна отделная статья, да и они нужны в основном только если ты захочешь написать свой бутлоадер. Про них все написано в даташите, а если не шаришь в аглицком, то книга Евстифеева [1] тебе будет хорошим подспорьем. Плюс есть куча примеров на этот счет.

Выбор bootloader'а
Мы же пока будем юзать готовый, благо их понаписано вагон и маленькая тележка. Но какой из них выбрать? Для себя я обозначил ряд критериев исходя из которых выбирал подходящий бут. Итак, вот они:

- Должен быть написан на Си (Бейсике, Паскале — нужное подчеркнуть). При всей моей любви к ассемблеру тут однозначно рулит Си. Дело в том, что удобно постоянно юзать один и тот же код, просто перекомпилируюя его под свой процессор. С ассемблером это довольно затруднительно, даже несмотря на единое ядро у AVR возникает куча различий на уровне периферии. Например, в ATmega8 регистр UART зовется как UDR, а в ATmega88 его обозвали уже UDR0 и обращаться к нему через команды IN/OUT уже нельзя — только через LDS/STS (т.к. из-за сильно разросшейся периферии его адресация вылезла за пределы досягаемости команды IN/OUT). Как результат — нативный ассемблерный код приходится править ручками. Хоть это и не сложно (обычно прокатывает автозамена), но не наш метод. Плюс ко всему, размер бутлоадера у нас меняется фиксированными кусками. А поскольку, например, в самый маленький сегмент полнофункциональный бутак все равно не влезет, хоть ты заизвращайся с кодом, в более толстый же сегмент уже без напряга влезает Сишный код. Так что экономии особой тут не нароешь.
- ; Должен поддерживаться родным софтом, в частности AVRProg'ом который идет в составе студии. Да, это немного громоздко, но зато не требует стороннего софта.
- Должен позволять писать как во флеш, так и в EEProm
- Должен быть не слишком жирным. Оптимально 512 слов. Меньше найдешь вряд ли, больше уже излишество.
- Должен быть под WinAVR (т.к. я юзаю именно этот компилер)

В результате был нарыт проект Martin Thomas'а из Германии «AVRPROG compatible boot-loader» [2] который идеально вписался в мои требования. Я лишь чуть чуть подправил его код под свои нужды.

И сейчас я покажу тебе как юзать эту замечательную программулину.

Итак, тебе нужна AVR Studio и WinAVR (она же AVR GCC). Если ничего этого ты еще не ставил, то сначала поставь студию, а потом сверху накати на нее WinAVR тогда они сцепятся друг с другом и будут работать в единой связке. **WinAVR** можно скачать с официального сайта [3]. Ставить лучше по дефолтному пути C:\WinAVR\ — меньше потом будет косяков с путями в либах.

Потом тебе нужны сорцы загрузчика. Можешь взять у у автора [2], но я рекомендую сдернуть все же у меня. У меня сразу же готовый проект для AVRStudio:

Скачать исходные коды и файлы проекта для Bootloader'а [4]
Я там чуток подправил для себя поэтому буду описывать свою версию. Ну и еще я некоторые комменты перевел, чтобы легче было ориентироваться в коде.

Компиляция
Запускай студию и создавай новый проект (если откроешь мой проект, то этот шаг можно пропустить) — **NewProject**. В качестве компилятора выбирай **AVR GCC**. Если опция **AVR GCC** не доступна, значит у тебя криво встала WinAVR, попробуй переустановить. В графу «Project Name» пиши что хочешь, а вот «**Initail File**» впиши «**main**» и поставь обе галочки что выше этой строки «**Create initial file**» и «**Create Folder**«.

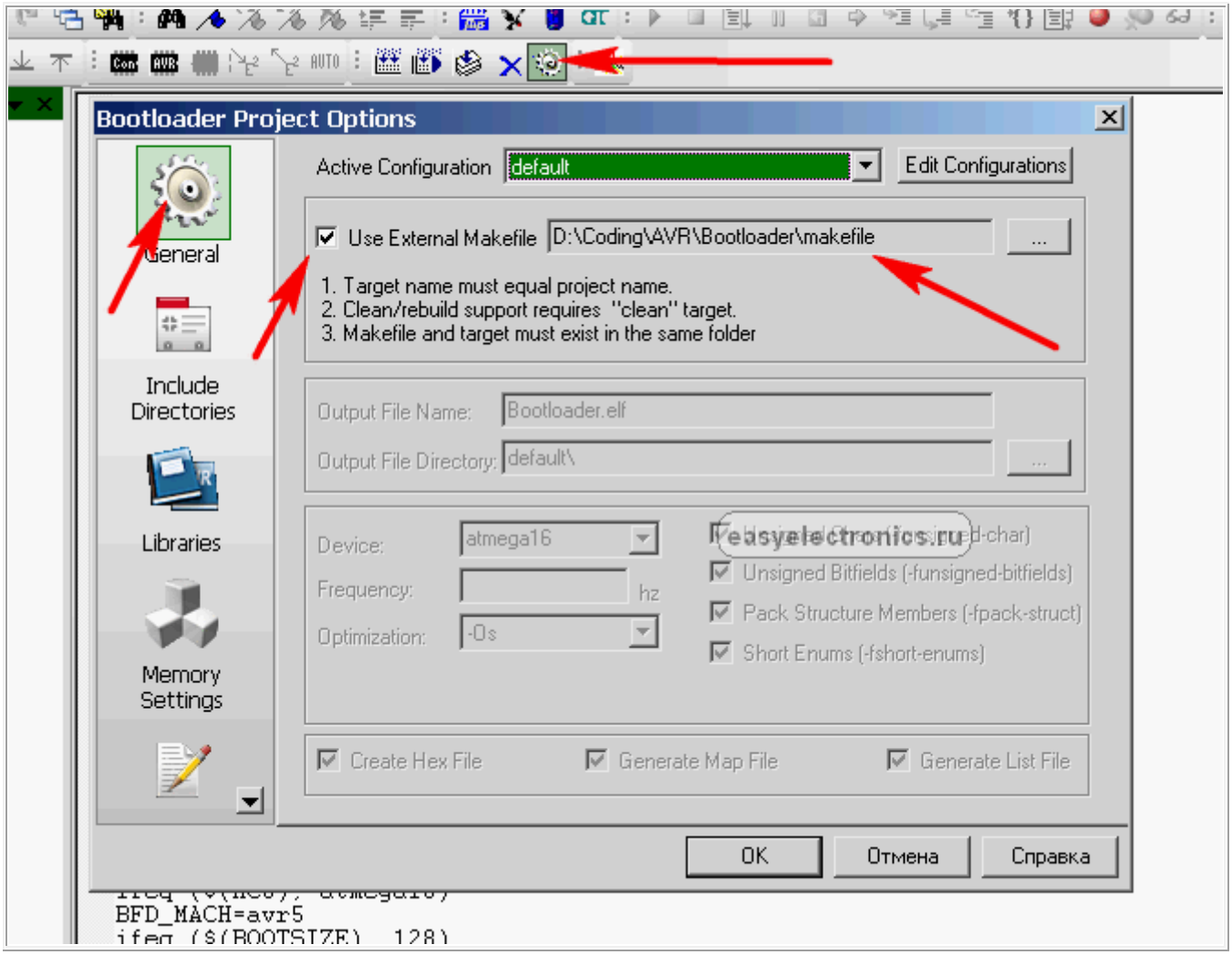
Следующим шагом выбирается микроконтроллер и отладочная платформа. В качетсве платформы выбери **AVR Simulator** и свою модель контроллера. Хотя, в принципе, это не так важно — всю работу сделает makefile. Но пусть будет.

Жмешь финиш и имеешь пустой проект. Сохраняешь его и закрываешь.

Теперь находишь папку этого проекта и перекидываешь туда все исходники и хидеры из архива бутлоадера, в том числе и папочку, что там тусуется, перезаписывая свой пустой файл **main.c** [5]

Осталось вновь открыть этот проект, но поскольку мы переписали main.c [5] то он уже не будет пуст. Там будет дофига кода.

Следующим шагом будет прописывание настроек компиляции. Поскольку тут уже есть готовый make то надо всего лишь подсунуть студии его. Тычь в иконку с шестеренкой и ставь в разделе General галку «Use External Make File» и прописывай путь к makefile который идет вместе с исходником.



Жми на компиляцию (F7) и фтыкай в сообщения об ошибках. По хорошему их не должно быть, разве что ругнется что не нашел elf файл. Но elf тебе никуда не уперся поэтому не заморачивайся. На эту ошибку можно забить.

Во вкладке Build AVR Studio должно проскочить что то вроде:

```
Build started 29.8.2009 at 20:22:36
-------- begin --------
avr-gcc (WinAVR 20090313) 4.3.2
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Size before:
main.elf  :
section          size       addr
```

```
.text           0x3ae     0x3c00
.trampolines      0x0     0x3fae
.bss             0x80   0x800060
.stab           0xeb8        0x0
.stabstr        0x9d4        0x0
Total          0x1cba
Size after:
main.elf  :
section           size       addr
.text           0x3ae     0x3c00
.trampolines      0x0     0x3fae
.bss             0x80   0x800060
.stab           0xeb8        0x0
.stabstr        0x9d4        0x0
Total          0x1cba
Errors: none
-------- end --------
Build succeeded with 0 Warnings...
```

Если полезли другие ошибки, например не найдена какая либо библиотека, то вполне возможно у тебя косяк в путях и придется прошерстить makefile и поправить пути к WinAVR и его библиотекам.

Конфигурация Bootloader'а
Но это так, тестовый прогон. Он тебе возможно нафиг не подойдет. Надо настроить лоадер под свой проц, свою частоту, свой размер.

Итак, вначале правим **main.c** [5]
Первым делом твой взор должен воткнуться в строку с определнием F_CPU:

```
1 #ifndef F_CPU
2 // #define F_CPU 7372800
3 //#define F_CPU (7372800/2)
4 #define F_CPU 8000000
5 //#define F_CPU 16000000
6 //#define F_CPU 12000000
7 #endif
```

Эта запись означает, что если у нас нигде не определана директива F_CPU, то вот тут мы эту самую директиву и назначим. И присваиваем ей тактовую частоту в герцах на которой будет работать наш проц. Это очень важно, ибо не настроишь частоту — не заработает правильно UART. Можешь просто раскомментировать нужную, а можешь добавить свою, не забыв закомментировать остальные.

Смотрим дальше

```
1 /* UART Скорость UART оптимально 19200 */
2 //#define BAUDRATE 9600
3 #define BAUDRATE 19200
4 //#define BAUDRATE 115200
5
6 /* Режим двойной скорости UART (бит U2C)*/
7 //#define UART_DOUBLESPEED
```

Это настройка скорости передачи порта по которому наш бут будет общаться с внешним миром. По дефолту идет 115200, но у меня на 115200 так и не заработало, поэтому я понизил до 19200. Работает стабильней некуда. Режим удвоения скорости я никогда не юзаю. Он пригождается лишь когда работаешь на медленных кварцах, вроде часового. А так — лишняя сущность. Поэтому закомменчено.

```
1  /*
2   * Выбор порта для кнопки входа в загрузчик
3   * Чтобы войти в загрузчик надо чтобы при запуске эта кнопка замыкала пин на землю
4   */
```

```
5  #define BLPORT          PORTD
6  #define BLDDR           DDRD
7  #define BLPIN           PIND
8  #define BLPNUM          PIND7
9
10 /*
11  * Выбор порта для индикатора работы загрузчика
12  * Светодиод горит - мы в загрузчике
13  */
14
15 #define ENABLE_BOOT_LED
16 #define BIPORT          PORTD
17 #define BIDDR           DDRD
18 #define BIPIN           PIND
19 #define BIPNUM          PIND5
```

Это выбор порта инициализации и порта индикации. Оба этих пина совершенно опциональные.
Порт инициализации нужен в том случае если мы в бутлоадер входим при замкнутом на землю выводе который вот этой настройкой и определяется. Я же обычно юзаю вход по таймеру, поэтому для меня эта опция не играет роли. А порт индикации это уже моя фича, сделана просто для удобства. На этот пин вешается светодиод и горит когда процессор находится в секции бутлоадера. Это позволяет мне знать что делает мой проц. Если фича не нужна, то закомментируй строку #define ENABLE_BOOT_LED и весь код обработки этого светодиода будет выпилен из исходника директивами условной компиляции ifdef-endif

Следующая важная опция — способ входа в бут

```
1  /*
2   * Выбор режима загрузчика
3   * SIMPLE-Mode - Загрузчик стартует когда нажата его кнопка
4   *   переход к основной программе осуществляется после сброса
5   *   (кнопка должна быть отжата) либо по команде от программатора
6   *   При этом режиме вывод на кнопку конфигурируется как вход-с подтягом,
7   *   но при выходе из загрузчика все выставляется по умолчанию
8   * POWERSAVE-Mode - Startup is separated in two loops
9   *   which makes power-saving a little easier if no firmware
10  *   is on the chip. Needs more memory
11  * BOOTICE-Mode - для зашивки  JTAGICE файла upgrade.ebn в Mega16.
12  *   что превращает ее в JTAG отладчик. Разумеется нужно добавить весь необходимый
13  *   обвяз на кристалл для этого. И частота должна быть везде прописана как 7372800
14  *   в F_CPU Для совместимости с родной прошивкой JTAG ICE
15  * WAIT-mode Bootloader ожидает команды на вход, если ее не было в течении промежутка
16 времени
17  *   (который настраивается) то проихсодит переход к основной программе.
18  */
19 //#define START_SIMPLE
20 #define START_WAIT
21 //#define START_POWERSAVE
22 //#define START_BOOTICE
23
24 /* Команда для входа в загрузчик в START_WAIT */
25 #define START_WAIT_UARTCHAR 'S'
26
27 /* Выдержка для START_WAIT mode ( t = WAIT_TIME * 10ms ) */
   #define WAIT_VALUE 400 /* сейчас: 300*10ms = 3000ms = 3sec */
```

Выбираешь нужную, а остальное комментируешь. В принципе, рулят START_SIMPLE — вход по наличию низкого уровня на заданом выводе и START_WAIT — вход в бутлоадер по спец символу из UART в течении времени которое определено в переменной WAIT_VALUE. У меня комп немного подтормаживает (старичок, ага) поэтому ставлю 4 секунды. Иначе AVRProg не успевает пнуть в порт символ и запустить бут. После 4х секунд начинается выполнение основной программы.

В этом файле закончили, переходим к **makefile** [6] тут комментарий начинается с #

Вначале выбираем свой проц:

```
1  # MCU name
2  ## MCU = atmega8
3  MCU = atmega16
4  ## MCU = atmega162
5  ## MCU = atmega169
6  ## MCU = atmega32
7  ## MCU = atmega324p
8  ## MCU = atmega64
9  ## MCU = atmega644
10 ## MCU = atmega644p
11 ## MCU = atmega128
12 ## MCU = at90can128
```

Затем размер бутсектора

```
1 #/* Select Boot Size in Words (select one, comment out the others) */
2 ## NO! BOOTSIZE=128
3 ## NO! BOOTSIZE=256
4 BOOTSIZE=512
5 ## BOOTSIZE=1024
6 ## BOOTSIZE=2048
```

Первые две опции не катят, ибо бут в них не влезет. А вот 512 и 1024 вполне пригодны. Для начала поставь 512, если не влезет, то изменишь на 1024. Размер бутлоадера зависит от количества включеных фишек и опции запуска. Для Wait и Simple хватает и 512 слов.

Ниже можешь посмотреть адреса загрузочных секторов у разных микроконтроллеров. Если твоего нет в списке, то можешь его добавить по аналогии, прописав по даташиту числа. Не забыв добавить его строку и в записях что я приводил выше. Еще надо поковырять файл chipdef.h который идет в составе сорцев и добавить и там свой контроллер, точнее его инклюдник.

Если поковырять makefile еще ниже, то можно найти где прописываются пути к WinAVR

```
1 # ----------------------------------------------------------------------------
2 # Define directories, if needed.
3 #DIRAVR = c:/winavr
4 #DIRAVRBIN = $(DIRAVR)/bin
5 #DIRAVRUTILS = $(DIRAVR)/utils/bin
6 #DIRINC = .
7 #DIRLIB = $(DIRAVR)/avr/lib
```

Это на случай если будут ошибки при компиляции.

Теперь сохраняй изменения в **makefile** [6] и залезь в папку проекта, и грохни там все *.hex *.o *.map *.lss.
Я не знаю почему, но видимо в одном из этих файлов Студия при предыдущей компиляции сохраняет параметры из make и дальнейшие правки makefile не приносят результата. Я минут 20 пытался понять какого черта у меня бутлоадер не хочет влезать в память, пока не заглянул в hex файл и не увидел, что адрес бутсектора начинается далеко за пределами памяти Меги16 — т.к. первый раз, для пробы, я скомпилировал под Мегу32 и не смотря на то, что в makefile я все поправил на Mega16 и сохранил, но при компиляции Студия настойчиво совала бут черти куда. Пока не удалил эти файлы и не скомпилил заново (они появляются при компиляции) ничего не заработало.

Теперь компилируй загрузчик и получай на выходе main.hex файл — обычную прошивку. Для проверки открой его блокнотом и позырь на начало и конец, предварительно рекомендую покурить формат Intel HEX [7]:

:10**3C00**0011241FBECFE5D4E0DEBFCDBF10E0A0E69B
……
тут куча барахла — это код нашего загрузчика
……
:0E**3FA0**;00B1CE5D9BFECF2CB8ADCEF894FFCF16

:0400000300003C00BD
:00000001FF

Жирным я выделил адреса которые тебя интересуют. **3C00** — адрес начала сектора (в байтах! В даташите он приведен в словах, так что умножай на два 1E00*2=3C00) размером в 512 слов. **3FA0** адрес начала последней строки кода загрузчика, а число перед адресом (0E) — длинна этой строки.
3FA0+0E = 3FAEадрес самого последнего байта загрузчика. Последний адрес флеш памяти для **ATmega16** это **3FFF** так что у нас еще 81 байт в запасе остался ;)))))

Ок, загрузчик с включенными фичами входит в память. Так что все в порядке. Если у тебя с твоим набором функций не влезет в память, то придется выбирать бутсектор на 1024 слова и перекомпилировать все заново.

Готово, у нас есть hex файл, можно прошивать. Подключаем программатор (да, он тут потребуется, а ты думал в сказку попал? Нифига, программатор штука такая без которой никуда) и заливаешь ее в МК. Сразу предупреждаю шиться будет долго. Т.к. будет заливаться полная прошивка на 16кБ (для мега16). Причем что **avrdude@usbasp**, что **AVRProg@JtagICE**— все ругаются на какую то ошибку, дудка порой виснет на чтении, однако если прочитать потом кристалл и сравнить два хекса, то будет видно, что лоадер четко встал в вверенные ему адреса.

Осталось сделать страшное :) Выставить Fuse биты.

Во-первых надо активировать бит **BOOTRST**
Во-вторых выставить размер бут сектора в битах **BOOTSZ1..0**, для 512 на **Mega16** это 01

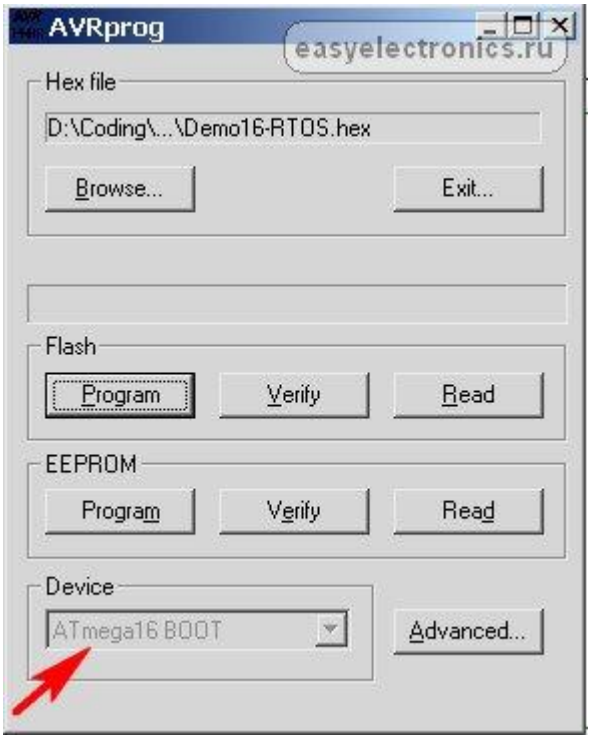
Готово.

Как проверить работу Bootloader'а
Теперь нам надо соединить микроконтроллер с компом через RS232 или его USB эмуляцию. [8] У меня USB, главное чтобы виртуальный USB был в числе первых четырех COM портов. У меня он зарегился в системе как COM4

Запускем терминалку, делаем коннект на порт к которому подключен наш контроллер и вызваем срабатывание условиях. У меня в данный момент бут настроен на Start_Wait поэтому я просто жму RESET (зажигается диод что проц в буте) и посылаю микроконтроллеру букву «S» в ответ мне приходит «AVRBOOT» Ок, работаем. Если ничего не делать, то диод погаснет через 4ре секунды — вышли из бут сектора. Но если в микроконтроллер не зашито еще ничего, то быстро проскочив через 16кб пустоты проц опять выполнит код бутлоадера и так по кругу, диод будет тупо гореть.

Как прошивать через Bootloader
Закрываем терминалку, чтобы освободить порт.
Подключаем контроллер, подаем питание. Запускаю студию, жму RESET на плате микроконтроллера и в меню студии быстро выбираю Tool — AVR Prog… Помним, что у нас в запасе 4 секунды.

Запускается AVRProg, стучится в порты с вопросом»S»?, а из одного из них ему «Алоха браза я тут!» Это радостное событие отмечается открытием окна AVR Prog:



Ну, а дальше тривиально. Выбираем кекс для флеша, если надо и для епрома и шьем. Шьется просто реактивно! Закрываем AVR Prog, делаем RESET, ждем 4ре секунды — прога пошла!

Ссылки по теме:
Загрузчик MegaLoad более компактный, но требует свою прогу [9]

3.Ы.
Народ, кто юзает/пишет загрузчики накидайте мне в комменты линков на то чем пользуетесь вы. А то я особо тему не рыл, нашел что удовлетворяло моим условиям и успокоился. Хотелось бы в статью еще добавить подборку линков на разные бутлоадеры под разные языки и компиляторы.

3.3.Ы
Подумалось тут…
Бутлоадер настроен на определенную частоту. Но для отладочной платы это не очень удобно, там порой может потребоваться смена частоты. Частоту можно изменить переткнув кварц, но тогда бутлоадер работать перестанет — собьется частота UART и потребуется опять программатор (которого может и не быть под рукой), чтобы перешить бутлоадер под новую частоту. Единственно что можно менять кварцы с частотами степени 2, тогда просто у USART будет скакать скорость, скажем, кварц на 8мгц — скорость 9600 бод, поставили кварц на 16мгц — скорость на том же буте стала уже 19200.

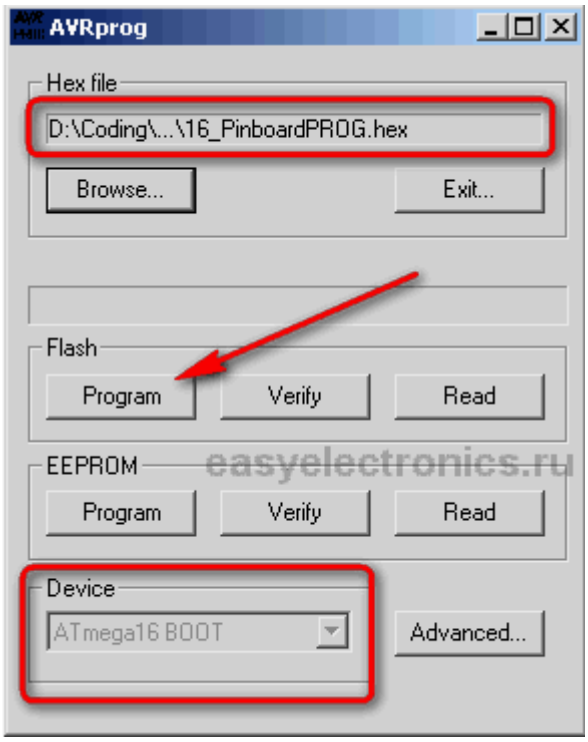
Проблема может решаться таким образом — числа определяющие скорость уарта мы кладем в последниие адреса EEPROM (если там 0, то можно задать какой нибудь дефолт, например 9600 для частоты 8Мгц, чтобы можно было хоть как то оживить МК), а затем, если мы хотим поменять кварц и перешить прогу на новую частоту, то сначала, на прежнем буте не перезагружаясь, пока он еще доступен, зашиваем вначале новую прошивку, потом новый Епром с числами под новый кварц, перезагружаемся — меняем кварц и опа — у нас опять МК онлайн. Надо добавить в этот лоадер такую фичу. =)

Прошивка PinboardProg для превращения демоплаты PinBoard в ISP программатор

Хай Pinboard [1] сообщество! Нас теперь уже почти сто человек :) Помнится я обещал, что будет прошивка позволяющая превратить демоплату в программатор для прошивки других МК. Пацан сказал — пацан сделал :)

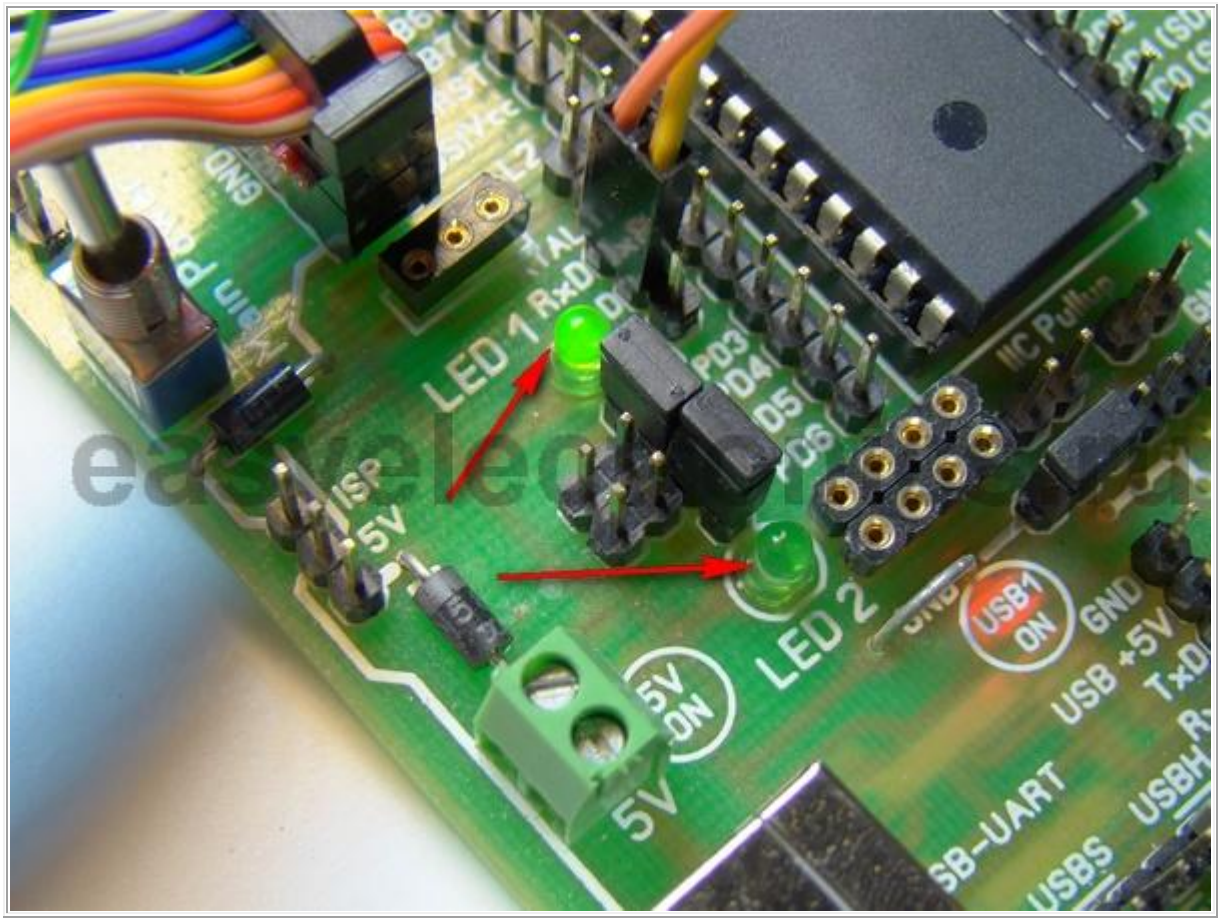
Итак, вот две прошивки. Для плат на базе Atmega16 и Atmega32. Тактовая 8 МГц (дефолтная настройка).
16_PinboardPROG.hex [2]
32_PinboardPROG.hex [3]

Далее как обычно в картинках.
Вначале стандартно, пинаем бутлодыря. И через AVRProg заливаем в память прошивку:



Прога сама определила что у нас бутлоадер, поэтому процессор выбрать нельзя.

После сего экшена закрываем AVRProg и перегружаем процессор платы. После отработки бута LED2 должен погаснуть (а на Мега32 они так не горит), но начнут перемигиваться диоды LED2 и LED1, чтобы это увидеть не забудь накинуть два джампера на выводы:

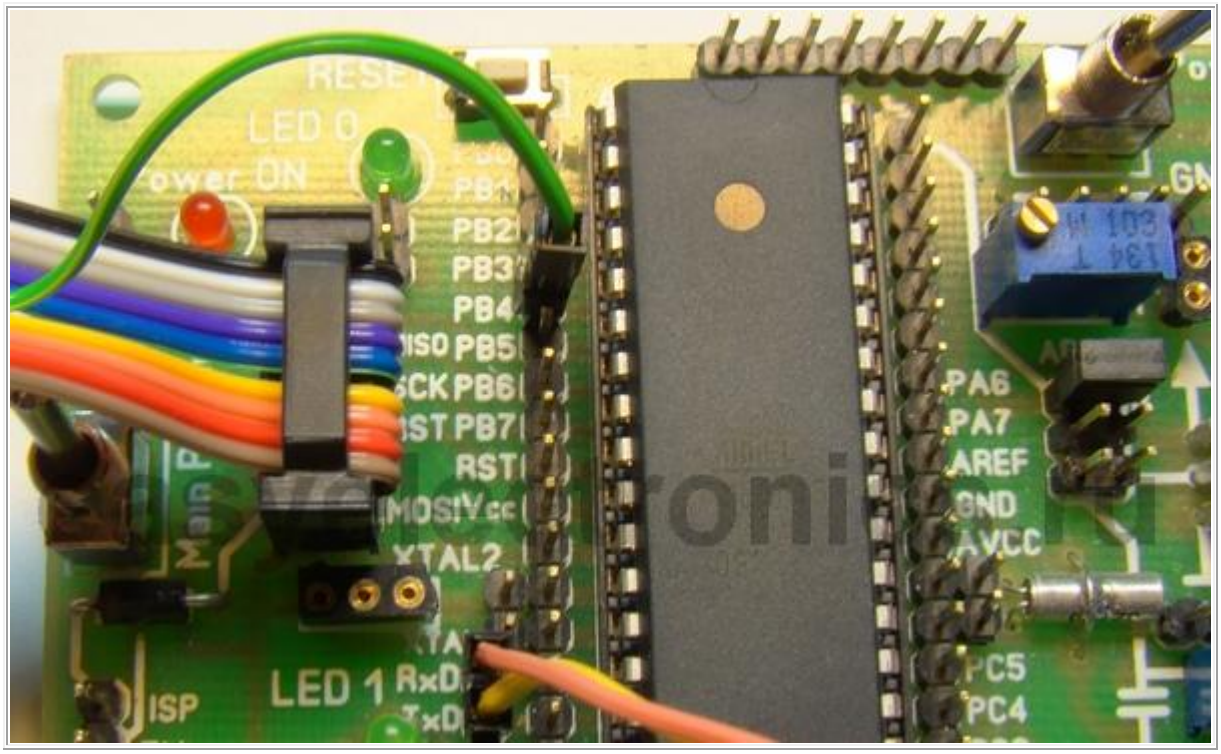


Если все замигало, значит теперь наша плата может шить. Но нам потребуется шнурок. Почти стандартный ISP кабелек. Отрубай от куска психоделичного радужного шлейфа сколько не жалко и насаживай IDC разьемы. Стрелочка показывает на метку первого контакта:

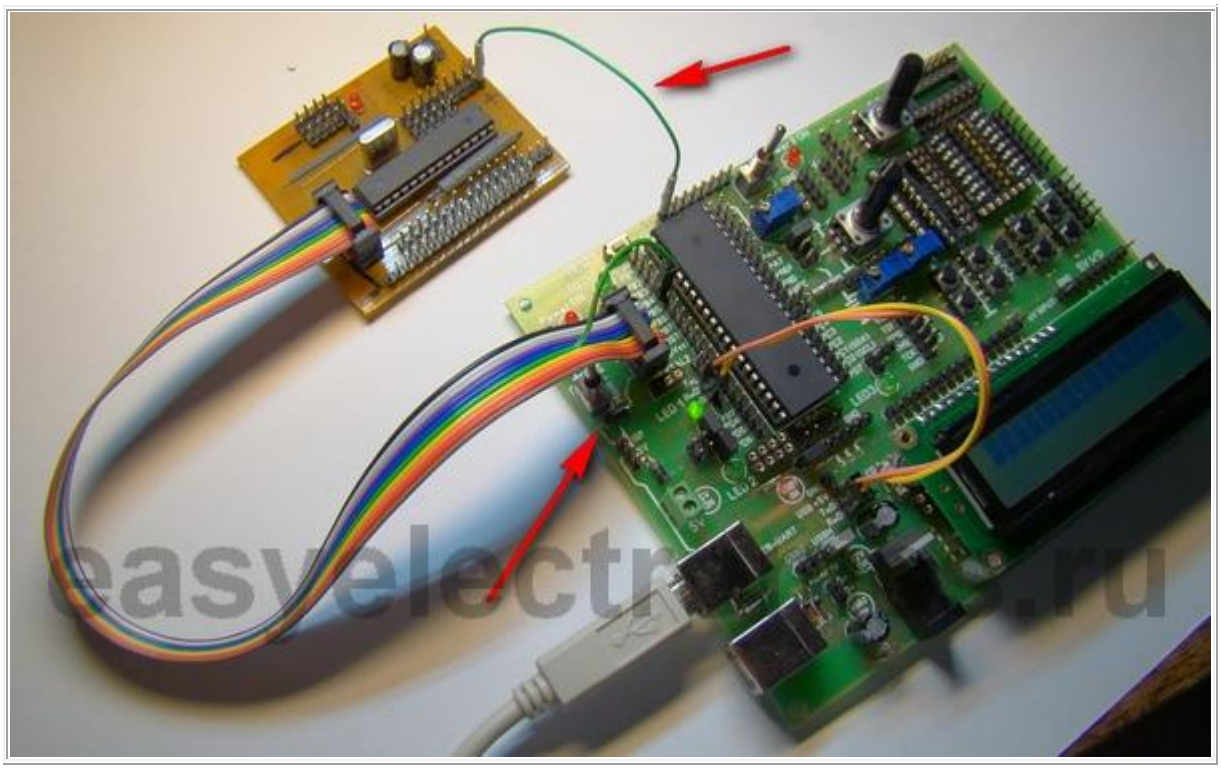
Один конец обжимается как обычно (обжимать их удобней всего маленькими тисами. Мне доводилось также для этих целей использовать дверь, а также крышку от унитаза. В общем, кто во что горазд. Но сразу не рекомендую плоскогубцы — большой риск расколоть нафиг), а вот второй обжимается хитрей.

Тут надо вычленить пятый проводок от метки. Метка на IDC возле ключевого язычка и обозначается треугольником. Пятый проводок это Reset.

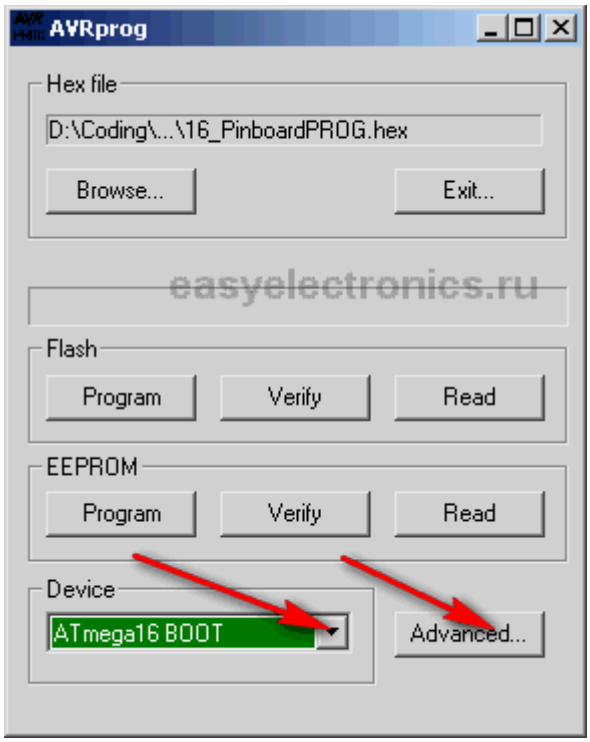
Нам ведь не нужно обьединять RESET шьющего и прошиваемого контроллеров, а надо снять его сигнал с вывода PB4 — он будет контроллировать прошивку. Соединяем так:



А вот тут лажа, IDC шки то я в комплекте не предусмотрел :(Хорошо, в дальнейших поставках учту. Впрочем, стоят они копейки и довольно часто встречаются. Но если нету, то ничего, можно и обычными контактами, что шли в комплекте. Соединяешь PinBoard с прошиваемым в целевой плате МК таким образом: MISO-MISO, MOSI-MOSI, SCK-SCK, GND-GND, RESET-PB4. Питание на Vcc берешь либо откуда нибудь извне, либо с шины Power, что над процессором. По идее оно может и по шлейфу на целевую плату уползти, но вот только защитный диод (возле тумблера который, на фотке стрелочкой показан) не даст. Но ведь никто не мешает его коротнуть на время ;) , главное не забыть поставить перемычки ISP и USBH одновременно (или от чего ты там плату запитал), чтобы собрать цепь питания.



Врубаешь питание. Ждешь пять секунд пока бут отработает, погаснет LED2 и зажгется LED1. Снова запускай AVRProg, как обычно. Но на этот раз его встретит не AVRBOOT, а AVR ISP =)



Видишь, можно выбирать тип МК, а также доступны всякие Fuse биты под кнопочкой адвансед. Выбираешь нужный тебе тип МК, выбираешь нужную прошивку. Шьешь!

По быстрому проверял, вроде фурчит. Шил мегу32, мегу16 и мегу8. Других не оказалось под рукой. Тестим! Да, с фузами пока осторожней играйтесь, кто его знает как оно себя поведет. Я вроде косяков не обнаружил, но мало ли.

Если вдруг не заработает, то проверять программатор так же как и бутлоадер. Т.е. коннектишься в терминалке. Вначале, пока работает бут, скармливаешь ему на скорости 19200 букву S должен ответить AVRBOOT. Потом сбрасываешь, ждешь пока бут загрузит основную программу и на той же скорости говоришь ему S, должен уже отозваться как **AVR ISP**

Прошива собрана на базе древнего, но тем не менее не забытого немецкого проекта [4]. Я просто переписал прошивку с AT90S2313 на Mega16/Mega32, да избавился от внешних кварцев.
Вот оригинальный сырок [5].

AVR. Учебный Курс. Конфигурация FUSE бит

В прошлых статьях я советовал тебе не лезть к этим битам. И на это были свои основания, так как неправильно выставив эти биты ты можешь наглухо заблокировать контроллер для дальнейшей перепрошивки или вообще какого либо использования.

Но без знания этой особенности контроллера далеко не уедешь. Так что распишу все по порядку. У разных версий контроллеров число FUSES разное, какие то могут отсутствовать, но основные есть всегда. Вот по ним и пройдемся.

Конфигурационные биты находятся в особой области памяти и могут быть изменены только с помощью программатора при записи контроллера. Есть старший байт и младший байт. Младший байт обычно отвечает за частоту, а старший за всякие фенечки.

Итак, главное:

В Atmel AVR принята следующая нотация: сброшенный в ноль fuse bit считается активным, т.е. включенным.

Пример Бит RSTDSBL, как можно догадаться из названия, это RESET DISABLE. Включил эту опцию и у тебя нога RESET превращается в порт ввода-вывода, но за это ты теряешь возможность перешить контроллер через ISP.

Так вот, чтобы выключить RESET (и получить большое западло с прошивкой в обмен на мелкую подачку в виде дополнительной ножки) в этот бит надо записать 0.

С одной стороны нелогично и криво. Как бы во всем мире принята нотация, что ноль это выключено, а тут, понимаешь, наоборот. С другой стороны, это их контроллер, что хотят то и делают. Один раз запомнить и все. Да и вообще, в электронике часто за сигнал берут ноль.

Однако контроллеры делают электронщики, а прошивающие программы — программисты. Как бы логично. И вот эти программисты взяли и заварили адскую путаницу с галочками. Нет бы им раз и навсегда принять за стандарт, что галочка это 1, а не ВКЛЮЧЕНО (что, напомню, является нулем). И поэтому в одних прошивающих программах галочка означает, что опция включена (в FUSE бит записывается 0),в других, обычно написанных электронщиками, галочка означает единицу. Т.е. с точностью до наоборот.

А что будет если перепутать? А будет ОЧЕНЬ плохо. Контроллер войдет в неправильный режим и может заблокируется наглухо. Т.е. раз прошил и все. Приехал.

Нет, спасти его можно, но для этого тебе потребуются дополнительные ухищрения в виде высоковольтного программатора, JTAG адаптера или генератора тактов. Все зависит от того в какой режим ты загонишь контроллер своими неправильными настройками.

Новичку, обычно, бывает проще сходить и купить новый МК, чем оживить заблокированный. Но не спеши отправлять его в помойку. Пометь и отложи на будущее, разберешься оживишь.

Конфигурация тактового сигнала
По умолчанию все контроллеры AVR (кроме старых серий AT90S2313, AT90S8535 итд) сконфигурированы так, чтобы работать от внутреннего источника тактов. Т.е. стоить подать на них питание и они начинают работать. Ничего больше и не нужно.

За источник тактов отвечают биты **CKSEL**
Выставив их правильным образом можно выбрать частоту работы контроллера, а также источник тактового сигнала.

- CKSEL3…0 = 0000 — Внешний источник сигнала.

Т.е. на вход XTAL1 подаются прямоугольные импульсы. Такое иногда делают в синхронных системах, когда несколько контроллеров работают от одного генератора.

Техническое отступление
В этот режим часто попадают, когда пытаются выставить контроллер на работу от внешнего кварца (CKSEL=1111), но либо путают нотацию, либо из-за прикола с обратной нотацией битов во всяких извратских прошивающих программах. Раз и кристалл заблокировался. Но, на самом деле, наглухо, с помощью CKSEL, заблокировать кристалл нельзя. Обычно все решается напайкой кварца и запуском от этого кварца. Худшее же что может случиться — потребуется внешний генератор тактов. Который бы оживил кристалл. Делается он за пять минут из любой микросхемы ТТЛ логики, например из К155ЛА3 — схем в инете навалом. Или на таймере 555, либо можно взять второй МК и на нем написать простую программку, дрыгающую ножкой. А если есть осциллограф, то с него можно поиметь сигнал контрольного генератора — его клемма должна быть на любом осциле. Землю осцила на землю контроллера, а выход генератора на XTAL1.

Но что делать если зуд нестерпимый, контроллер залочен, а никакой микросхемы для реанимации под рукой нету? Тут иногда прокатывает метод пальца. Прикол в том, что на тело человека наводится весьма нефиговая наводка частотой примерно 50Гц. Всякий кто хватался за щупы осциллографа руками помнит какие шняги тут же возникают на экране — вот это оно! А почему бы эту наводку не заюзать как тактовый сигнал? Так что припаиваешь к выводу XTAL1 провод, хватаешься за него рукой, и жмешь на чтение или запись контроллера :) Предупреждаю сразу, метод работает через жопу, далеко не с первого раза, читает долго и порой с ошибками, но на перезапись FUSE битов в нужную сторону должно хватить. Пару раз у меня такой фокус получался.

CKSEL3…0 = 0100 – 8 MHz от внутреннего генератора(обычно по умолчанию стоят такие)
Для большинства AVR такая конфигурация CKSEL означает тактовку от внутреннего генератора на 8Мгц, но тут

могут быть варианты. Так что в этом случае втыкай внимательно в даташит. В табличку Internal Calibrated RC Oscillator Operating Modes

Иногда нужно иметь внешний тактовый генератор, например, чтобы его можно было подстраивать без вмешательства в прошивку. Для этого можно подключить RC цепочку, как показано на схеме и подсчитать частоту по формуле **f = 1/3RC**, где f будет частотой в герцах, а R и C соответственно сопротивлением резистора и емкостью конденсатора, в омах и фарадах.

- CKSEL3…0 = 0101 – для частот ниже 0.9 MHz
- CKSEL3…0 = 0110 – от 0.9 до 3 MHz
- CKSEL3…0 = 0111 – от 3 до 8 MHz
- CKSEL3…0 = 1000 – от 8 до 12 MHz

Данная табличка справедлива только для ATmega16 у других МК может отличаться. Уточняй в даташите!

Проблема у внутреннего генератора и внешних RC цепочек обычно в нестабильности частоты, а значит если сделать на ней часы, то они будут врать, не сильно, но будут. Поэтому иногда полезно запустить контроллер на кварце, кроме того, только на кварце можно выдать максимум частоты, а значит и производительности проца.

- CKSEL3…0 = 1001 — низкочастотный «часовой» кварц.

На несколько десятков килогерц.
Используется в низкоскоростных устройствах, особенно когда требуется точная работа и низкое потребление энергии.

Для обычных кварцев ситуация несколько иная. Тут максимальная частота кварца зависит также и от бита **CKOPT** когда CKOPT = 1 то:

- CKSEL3…0 = 1010 или 1011 — от 0,4 до 0.9 MHz
- CKSEL3…0 = 1100 или 1101 — от 0,9 до 3 MHz
- CKSEL3…0 = 1110 или 1111 – от 3 до 8 MHz (либо от 1 до 16Мгц при CKOPT=0)

А если **CKOPT** равен 0 то при тех же значения CKSEL можно поставить кварц от 1 до 16MHz.

Разумеется, кварц на 16MHz можно поставить только на Мегу без индекса ”L”. (Хотя, как показывает практика, Lky тоже можно неслабо разогнать. У меня ATMega8535L заработала на 16Мгц, но были странные эффекты в работе. Поэтому я не стал так извращаться и разгон снял). Опять же, все выше сказанное в точности соответствует только Меге 16, у других может незначительно отличаться.

Бит **CKOPT** задает размах тактового сигнала. Т.е. амплитуду колебаний на выходе с кварца. Когда CKOPT = 1 то размах маленький, за счет этого достигается меньшее энергопотребление, но снижается устройчивость к помехам, особенно на высоких скоростях (а предельной, судя по таблице выше, вообще достичь нельзя. Точнее запуститься то он может запустится, но вот надежность никто не гарантирует). А вот если CKOPT активизировать, записать в него 0, то размах сигнала сразу же станет от 0 до питания. Что увеличит энергопотребление, но повысит стойкость к помехам, а значит и предельную скорость. При оверклокинге МК тем более надо устанавливать CKOPT в 0.

Также стоит упомянуть бит SCKDIV8 которого нет в Atmega16, но который часто встречается в других контроллерах AVR. Это делитель тактовой частоты. Когда он установлен, т.е. в нуле, то частота выставленная в битах CKSEL0…3 делится на 8, на чем в свое время прилично застрял Длинный, долго пытаясь понять чего это у него западло не работает. Вся прелесть в том, что этот делитель можно отключить программно, записав в регистр CLKPR нужный коэффициент деления, например один. Весь прикол в том, что SCKDIV8 активен по дефолту! Так что внимательней!

Биты SUT задают скорость старта МК после снятия RESET или подачи питания. Величина там меняется от 4ms до 65ms. Мне, за всю практику, пока не довелось эту опцию использовать — незачем. Так что ставлю на максимум 65ms — надежней будет.

Бит **RSTDISBL** способен превратить линию **Reset** в одну из ножек порта, что порой очень нужно когда на какой-нибудь крошечной Tiny не хватает ножек на все задачи, но надо помнить, что если отрубить Reset то автоматически отваливается возможность прошивать контроллер по пяти проводкам. И для перешивки

потребуется высоковольтный параллельный программатор, который стоит несколько тысяч и на коленке сделать его проблематично, хотя и возможно.

Второй заподлянский бит это **SPIEN** если его поставить в 1, то у тебя тоже мгновенно отваливается возможность прошивать по простому пути и опять будет нужен параллельный программатор. Впрочем, успокаивает то, что сбросить его через SPI невозможно, по крайней мере в новых AVR (в старых, в AT90S*** было можно)

WDTON отвечает за Собачий таймер, он же Watch Dog. Этот таймер перезагружает процессор если его периодически не сбрасывать – профилактика зависаний. Если WDTON поставить в 0, то собаку нельзя будет выключить вообще.

BODLEVEL и **BODEN** — это режим контроля за напряжением. Дело в том, что при определенном пороге напряжения, ниже критического уровня, контроллер может начать сильно глючить. Самопроизвольно может запортачить, например, EEPROM или еще что откосить. Ну, а ты как думал, не покорми тебя с пару недель — тоже глючить начнешь :)
Так вот, для решения этой проблемы есть у AVR встроенный супервизор питания. Он следит, чтобы напруга была не ниже адекватного уровня. И если напруги не хватает, то просто прижимает RESET и не дает контроллеру стартовать. Вот эти два фуза и рулят этой фичей. **BODEN** включает, а **BODLEVEL** позволяет выбрать критический уровень, один из двух. Какие? Не буду раскрывать, посмотри в даташите (раздел System Control and Reset).

JTAGEN — Включить JTAG. По умолчанию активна. Т.е. JTAG включен. Из-за этого у MEGA16 (а также 32 и прочих, где есть JTAG) нельзя использовать вывода порта C, отвечающие за JTAG. Но зато можно подключать JTAG отладчик и с его помощью лезть контроллеру в мозги.

EESAVE — Защита EEPROM от стирания. Если эту штуку включить, то при полном сбросе МК не будет стерта зона EEPROM. Полезно, например, если в EEPROM записываются какие-либо ценные данные по ходу работы.

BOOTRST — перенос стартового вектора в область бутлоадера. Если эта галочка включена, то МК стартует не с адреса 00000, а с адреса бутсектора и вначале выполняет бутлоадер. Подробней про это было написано в статье про прошивку через лоадер.

BOOTSZ0..1 — группа битов определяющая размер бут сектора. Подробней смотри в даташите. От контроллера к контроллеру они отличаются.

Lock Bits
Это, собственно, и к фузам то отношения не имеет. Это биты защиты. Установка этих битов запрещает чтение из кристалла. Либо флеша, либо ЕЕПРОМА, либо и того и другого сразу. Нужно, только если ты продаешь свои устройства. Чтобы злые конкуренты не слили прошивку и не заказали в китае более 9000 клонов твоего девайса, оставив тебя без штанов. Опасности не представляют. Если ты заблокируешь ими кристалл, то выполни полное стирание и нет проблемы.

Характерной особенностью установленных лок битов является считываемая прошивка — в ней байты идут по порядку. Т.е. 00,01, 02, 03, 04… FF, 00… Видел такую срань? Значит не судьба тебе спереть прошивку — защищена =)

Техника безопасности
И главное правило при работе с FUSE битами — ВНИМАНИЕ, ВНИМАНИЕ и ЕЩЕ РАЗ ВНИМАНИЕ! Не выставляйте никогда FUSE не сверившись с даташитом, даже если срисовываете их из проверенного источника.

Мало ли в какой нотации указал их автор, в прямой или инверсной. Так что если повторяете какую-либо конструкцию, то перед тем как ставить фузы, проверьте то ли вы вообще ставите!

Обязательно разберитесь что означает галочка в прошивающей программе. Ноль или единицу. Включено или выключено! Стандарта нет!!!

Если фуз биты задаются двумя числами — старший и младший биты, то выставляются они как в даташите. Где 0 это включено.

Неплохой FUSE калькулятор [1]

Второе, железное, правило работы с FUSE. Запомните это навсегда и не говорите, что я не учил.

ВНАЧАЛЕ ЧИТАЕМ ТЕ ЧТО ЕСТЬ, ПОТОМ ЗАПИСЫВАЕМ ТЕ КОТОРЫЕ НАДО НАМ

Чтение — модификация — запись. ТОЛЬКО так. Почему? Да просто часто бывает как — открыл вкладку FUSE, а программатор попался тупой и сам их не считал. Как результат — у тебя там везде пустые клеточки. Довольный, выставил только те, что тебе надо SKSEL, а потом нажал WRITE. Тут то и наступает, Обычно, кабздец. Т.к. в контроллер записываются не только те, что ты изменишь, а ввобще вся секция. С теми самыми пустыми клеточками. Представь какой трешняк там будет. То то же! А потом бегут жаловаться по комментам и форумам, мол я ничего такого не трогал — оно само. Ага, щаз!

Так что, еще раз — Чтение, Модификация, Запись!

Подсказка:
Как с одного взгляда определить какого типа (прямые или инверсные) fuse биты в незнакомой прошивающей проге?
Дедуктивный метод: Нажмите чтение Fuses и посмотрите на состояние бита SPIEN Этот бит всегда активен, а если он будет сброшен, то программатор контроллер даже определить не сможет. Если SPIEN в 1 — значит фьюзы инверсные, как в PonyProg. Если ноль — значит по нотации Atmel.

Отладочная плата PinBoard v1.1

В один прекрасный момент я задумался — а какого черта я трачу столько времени на сооружении разного обвяза при отладке новых модулей и при подготовке экспериментов?

Ведь все можно сделать в единой отладочной плате, чтобы можно было просто соединить нужными перемычками блоки и получить сразу кусок решения. Нужна была демоплата.

Что такое демоплата? Демоплата — это универсальный полигон для экспериментов. Идеальное средство для быстрого старта. На ней смонтирован микроконтроллер, а также вся необходимая обвязка для его работа. Плюс разные полуфабрикаты, позволяющие облегчить эксперименты.

Готовое устройство, где все уже подключено, разведено как надо. Купил — включил — работай. Риск что-либо сжечь в ходе экспериментов снижен максимально. Где нет нужды парить мозг проблемой «чем прошить», «как запустить», «Как правильно все подключить». Первые шаги делаются быстро и легко.
А потом, когда ты уже освоишься с контроллером, то запросто можно посмотреть как сделано на демоплате и перенести это решение в свое устройство.

В принципе, идея не нова — по такому пути идет компания Микроэлектроника с их демоплатой EasyAVR5 (от 7 до 9 тысяч рублей). Плата замечательная, но на мой взгляд дороговата и многие фичи там избыточные и одноразовые. В том смысле, что раз побаловался, изучил и больше они не интересны. А кое чего нет вообще, например возможности по быстрому сварганить аналоговую цепочку или фильтр.

Плюс ко всему, в исходном варианте EasyAVR5 идет голой, только плата с простейшей периферией вроде светодиодов и кнопочек, а LCD дисплеи модули расширения надо покупать отдельно и стоят они тоже негуманно. Основную же стоимость добавляет многослойная плата с кучей джамперов и переключателей, позволяющая витиевато все это коммутировать.

Такой расклад меня не устроил поэтому я посидел несколько дней, поворошил в памяти все основные грабли и неудобства которые мне приходилось часто решать, попытался продумать какие задачи мне еще предстоит расковырять и что для этого неплохо было бы подготовить и получилось вот что:

Концепция была отличной от традиционных отладочных плат. Я не стремился до предела нафаршировать ее разной периферией, но постарался по максимуму облегчить и ускорить подключение к ней чего угодно.

Вот фичи которые мне удалось реализовать:
Независимость и взаимозаменяемость контроллеров
Микроконтроллер должен быть максимально независимым. То есть не должно быть такой ситуации, что вот эти вот ножки жестко завязаны на LCD дисплей, а эти на матрицу кнопок. Это можно легко развести и будет красиво смотреться, но когда переносишь в реальное устройство то куда лучше разводить плату как тебе удобно, а не как она у тебя разведена и запрограммирована!) на монтажной плате. Поэтому у меня есть независимые блоки, соединяющиеся длинными проводными перемычками в любом порядке.

Где это возможно я старался делать с помощью обычных комповых джамперов. Сам микроконтроллер может быть любым из сороконогих от ATMega8535 до ATmega32. По дефолту стоит Mega16.

Вся прелесть архитектуры AVR в том, что их микроконтроллеры различаются, по большей части, лишь количественно — больше/меньше ног, флеша, памяти, периферии. А сам код совместим.
Так что если надо перебросить код с ATMega16 на Atmega8 или ATTiny2313 то достаточно только поменять файл макроопределений *.inc А остальное останется почти также. Ну может заменить название некоторых битов и регистров, тут компилятор сам подсветит ошибку. Причем я говорю про код написанный на ассемблере !!!). Если же писать на Си, то править вообще скорей всего ничего не придется, только лишь указать другой тип МК.

Встроенный внутрисхемный программатор и гибкая система прошивки
Основная работа с платой идет через bootloader.
Это позволяет быстро зашить программу, а также полностью защищает от классических ошибок начинающих — блокировки кристалла неправильно выставленными FUSE битами. Посредством загрузчика нельзя что-либо испортить.

Также, зашив контроллер демоплаты специальной прошивкой ISP программатора, демоплата легко превращается в ISP программатор, способный прошить AVR контроллер другому независимое устройству, например вашей самостоятельной разработке. Для этого нужен всего лишь специальный шнур входит в комплект).

Если же вдруг потребуется подкорректировать FUSE биты процессора демоплаты или залить bootloader в новый процессор при модернизации платы на более мощный контроллер), то это не составит проблемы.

;

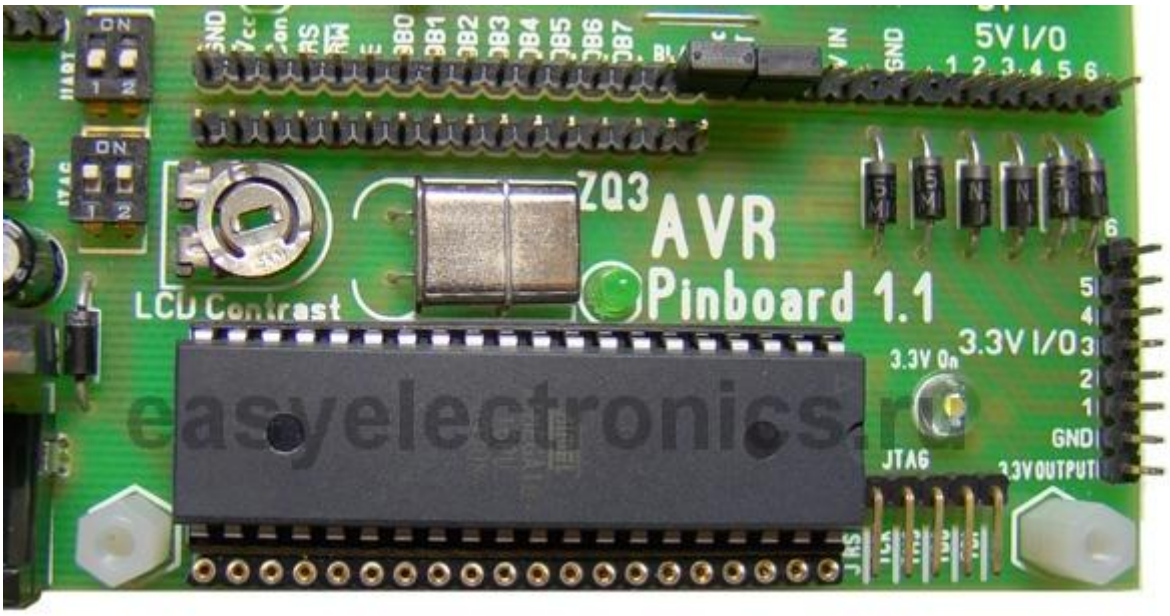
Ведь в плату встроен скоростной USB программатор, способный прошивать практически все виды контроллеров серии AVR.

Для работы с ним используется мощная и очень популярная консольная программа avrdude Windows,Linux,MacOS).

А также простая графическая оболочка GUI (SinaProg), на случай если вас смущает консольный интерфейс avrdude.

Также можно прошить прошить программу через интерфейс JTAG.

Внутрисхемная отладка
В плату встроен дополнительный микроконтроллер который может организовать внутрисхемную отладку по интерфейсу JTAG.



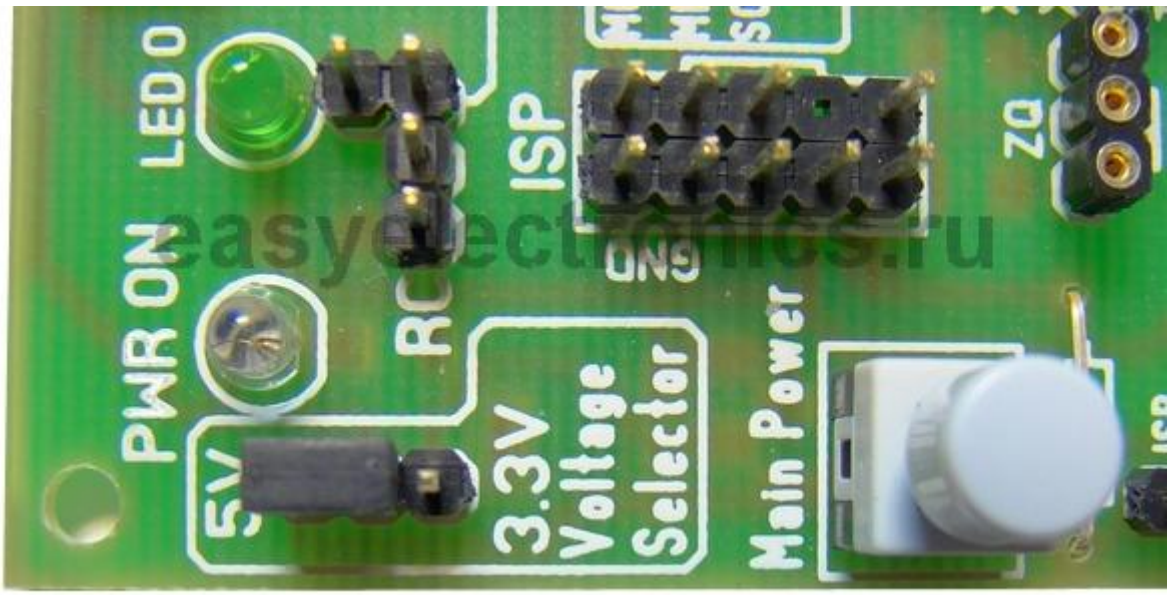
JTAG это мощнейший инструмент, позволяющий заглянуть в недра реально работающего контроллера. Найти ошибки в программе или разобраться в запутанной логике устройства.
С помощью JTAG можно:

- Пошагово выполнять программу в реальном контроллере.
- Смотреть состояние регистров, переменных, вручную их произвольно менять.
- Ставить точки останова на разные ключевые события.
- Прошивать микроконтроллер и менять fuse биты.

Отладка идет в AVR Studio — там же где и пишется программа.

При желании можно зашить в контроллер JTAG адптера любую другую программу и использовать ее как вспомогательный микроконтроллер. А когда нужда в дополнительном контроллере отпадет сменить прошивку обратно на JTAG.
Все прошивки и инструкции по перешивке контроллера прилагаются.

Возможность задать разное питающее напряжение для основной системы
Можно запитать всю систему как от 5 вольт, так и от 3.3 вольт.



Для чего на плате смонтирован дополнительный вводной преобразователь. Это бывает полезно при отладке низковольной аппаратуры.

Возможность задать любую частоту.
Отключаемые и сменные кварцы, наличие отключаемого часового кварца для реализации часов реального времени.

Упор на развитие интерфейсов.
SPI, I2C, USART — все выведено и сгруппировано так, чтобы можно было одним сплошным шлейфом утащить это на какой либо внешний модуль и не устраивать паутину из проводов.

Плюс подключаемый внешний обвяз для тех интерфейсов которые того требуют. Например, i2c требуют подтягивающих резисторов на шине. Воткнул пару джамперов и они уже есть.

Возможность питания из разных источников.
Чтобы без проблем можно было заставить питаться плату от USB шины, от блока питания на 9…12V, от грохотки батареек или ISP кабеля программатора.
Все это задается джамперами и с защитой от переполюсовки.

Раздельное управление питанием основного модуля и внешних подключаемых модулей.

Я сделал рубильник главного питания и рубильник вторичного питания, который отключает шину вторичного питания.

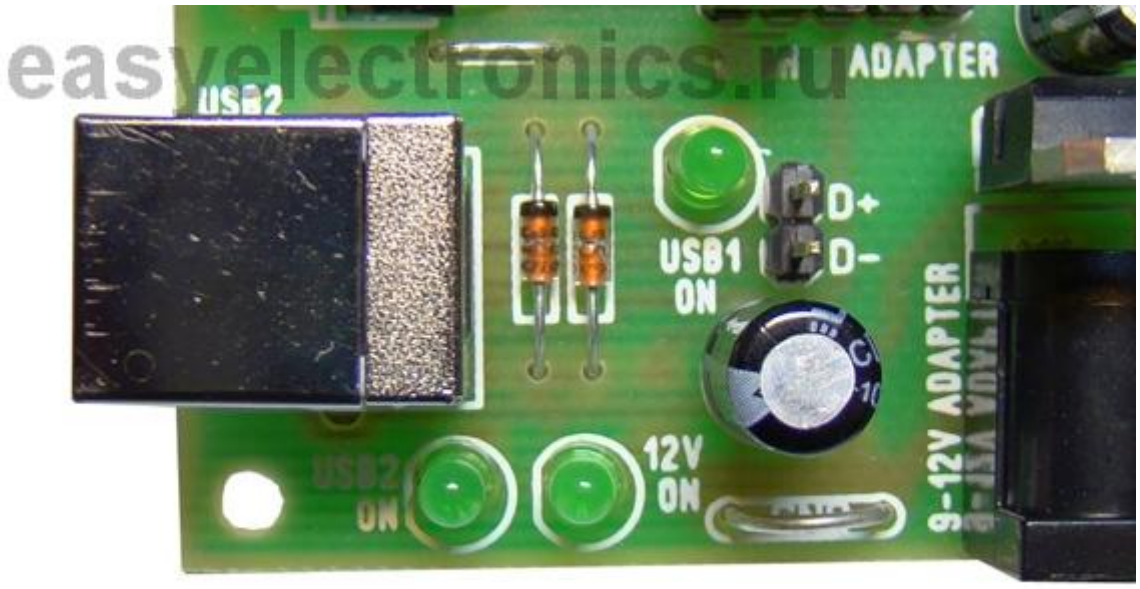
Удобно, например, когда надо чтобы МК работал, а внешняя периферия была обесточена. Щелк рубильничком и не надо дергать проводки. Или обесточил рубильником, а сам завел питалово на ту же шину с более мощного источника. В свое время мне этой возможности не хватало, поэтому добавил.

Есть маломощный источник напряжения на 3.3 вольта и 100мА для питания разной низковольтной аппаратуры. Например, дисплеи от сотовых телефонов работают на 3.3 вольтах.

С возможностью связи с компом.
Я сразу же заложился на микросхему преобразователь USB-UART FT232RL. Таким образом, я получил готовый интерфейс для связи с компом с надежными и стабильными драйверами, работающими как под виндой, так и под Линухи всякие. И не требующий редкого ныне COM порта.

Два щелчка дип переключателя и микроконтроллер готов к обмену данными с компьютером.
Опять же, следуя концепции изоляции МК, я не стал наглухо заводить терминал на выводы контроллера, а сделал их отключаемыми и вывел на штыри, что позволяет повесить терминалку куда угодно, не обязательно на МК.

Программный USB
Существует очень много проектов на программном USB драйвере от OBDEV. Чтобы не стоять в стороне от прогресса я добавил в свою плату второй USB с необходимым обвязом из резисторов и стабилитронов.

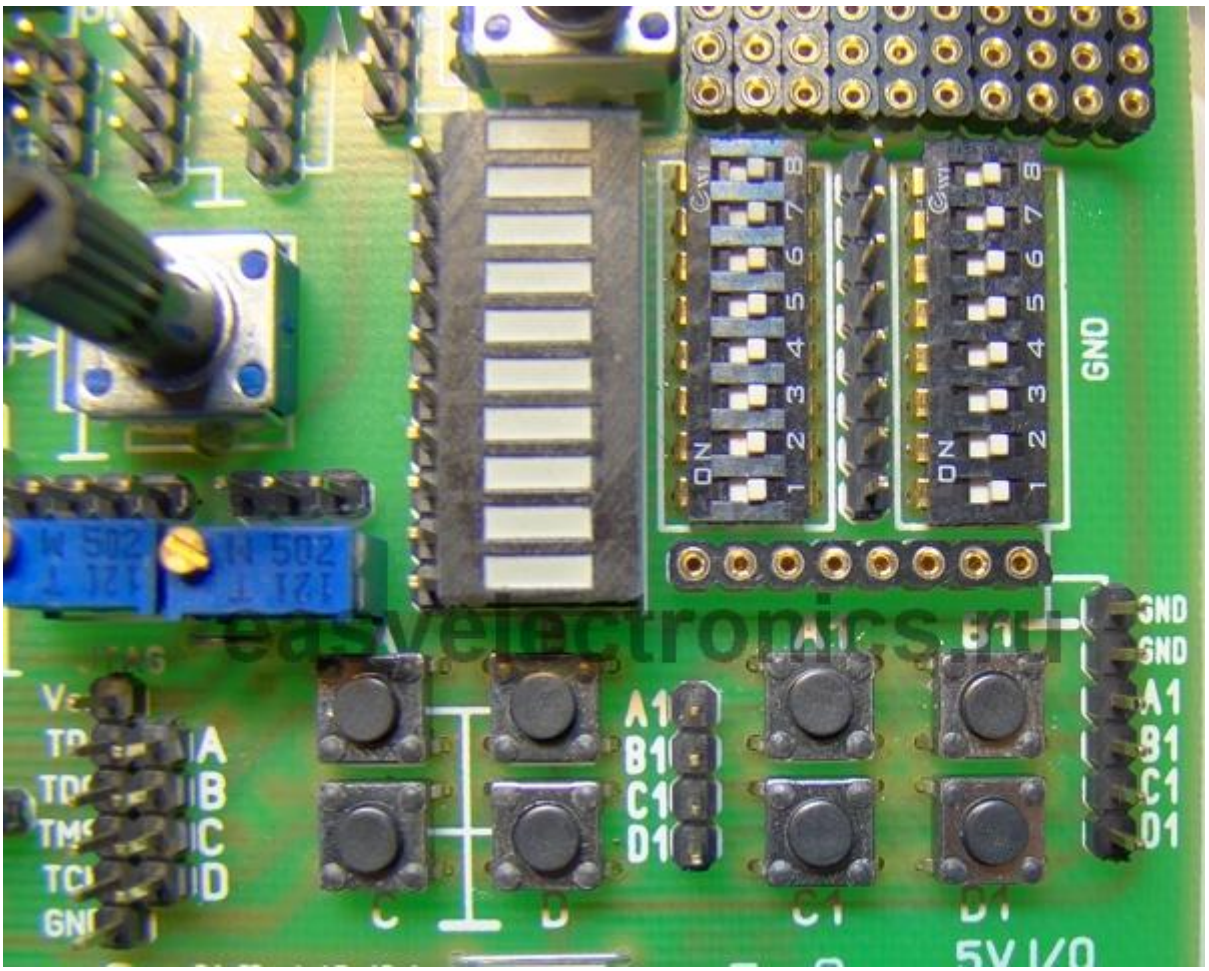


Так что если захочется реализовать в отладочной плате софтверный USB то достаточно будет просто бросить два проводка до контроллера.

Индикация
Наличие питания, причем соответствующим светодиодом показывается с какого источника это питание подведено.
Индикация передачи данных по последовательному порту.
Возможность в одно движение, не создавая паутины, джамером подключить четыре светодиода а так как они подключены к ШИМ, то можно и плавно управлять их яркостью).
Светодиодная линейка которую можно проводками прицепить куда угодно.
Плата комплектуется сьемным блоком с LCD на 2 строки 16 символов в каждой, на базе HD44780.

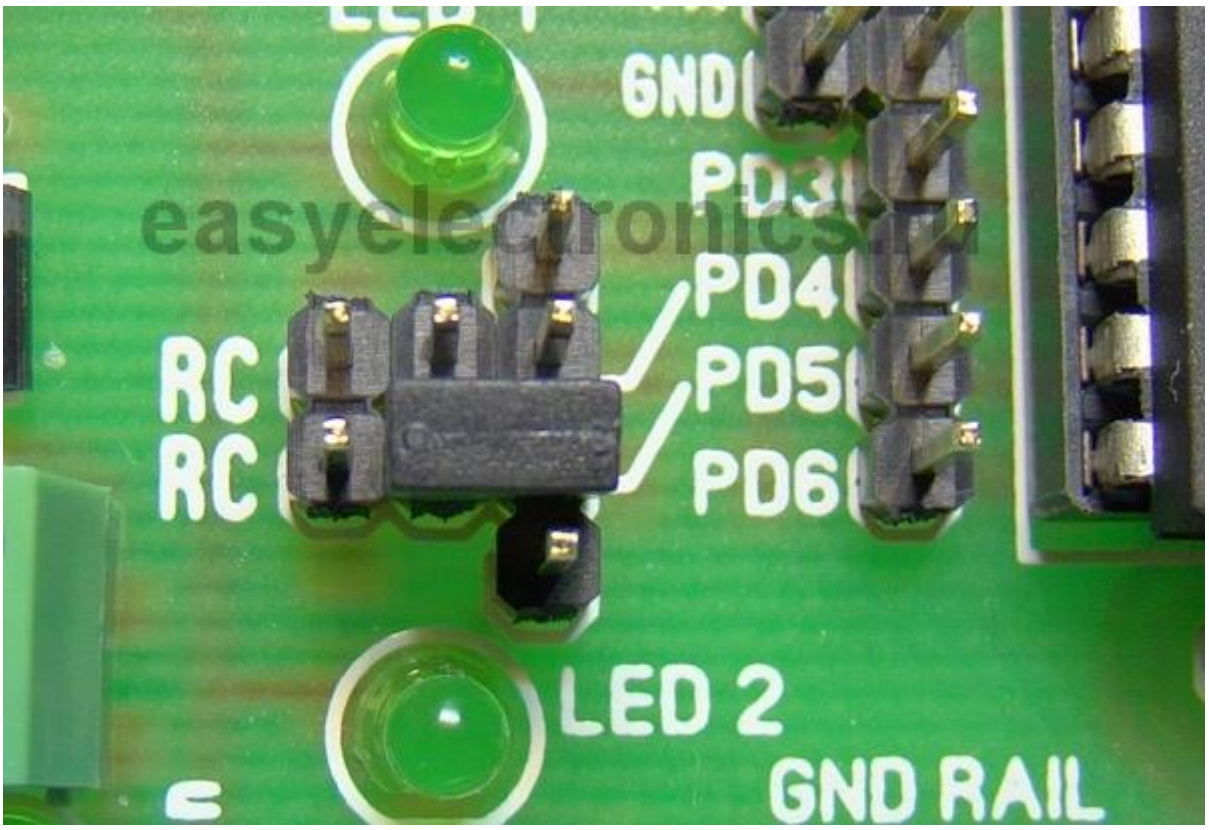
Гибкое управление подсветкой дисплея. Можно поставить джампер сразу на шину питания и включить дисплей на постоянную работу, а можно подать ШИМ сигнал с ноги микроконтроллера и управлять подсветкой плавно.

Управление
Четыре независимые кнопки с замыканием на землю
Четыре независимые кнопки с двумя свободными концами — можно подключить их на что угодно.



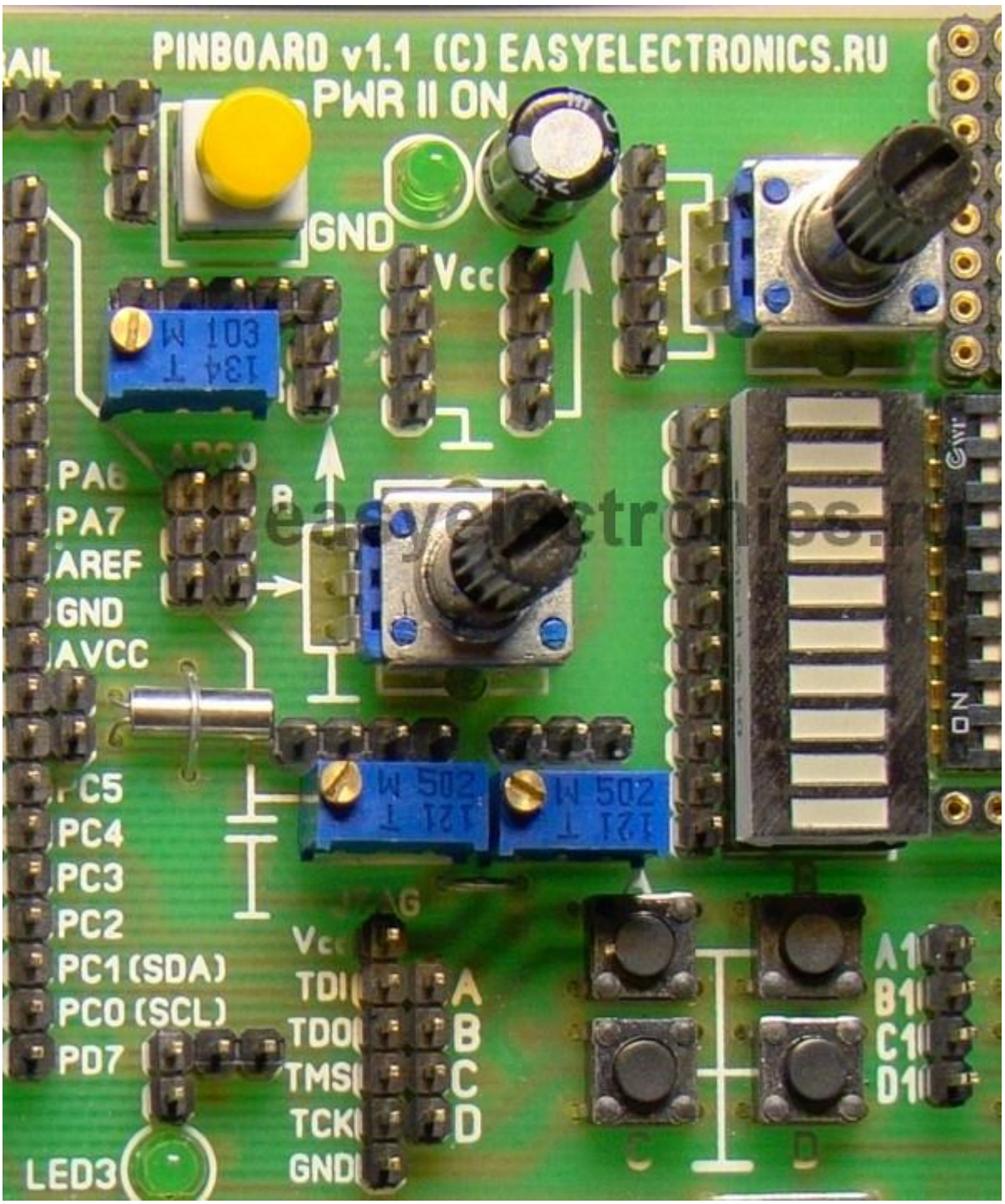
Восемь штырей состояние которых определяется DIP переключателями — может быть в режиме Hi-Z, PullUp 10k или GND. Очень удобно когда отлаживаешь сторонний модуль и надо на разные его входы подавать то единицу, то ноль, для входа в разные режимы. Да и вообще пригождается постоянно.

Простейший аналоговый обвяз
Для ШИМ часто нужен простейший RC фильтр чтобы из прямоугольного получить постоянное напряжение. У меня такой фильтр на любой из четырех ШИМ каналов подключается одним лишь джампером.



Раз и имеем фильтр.

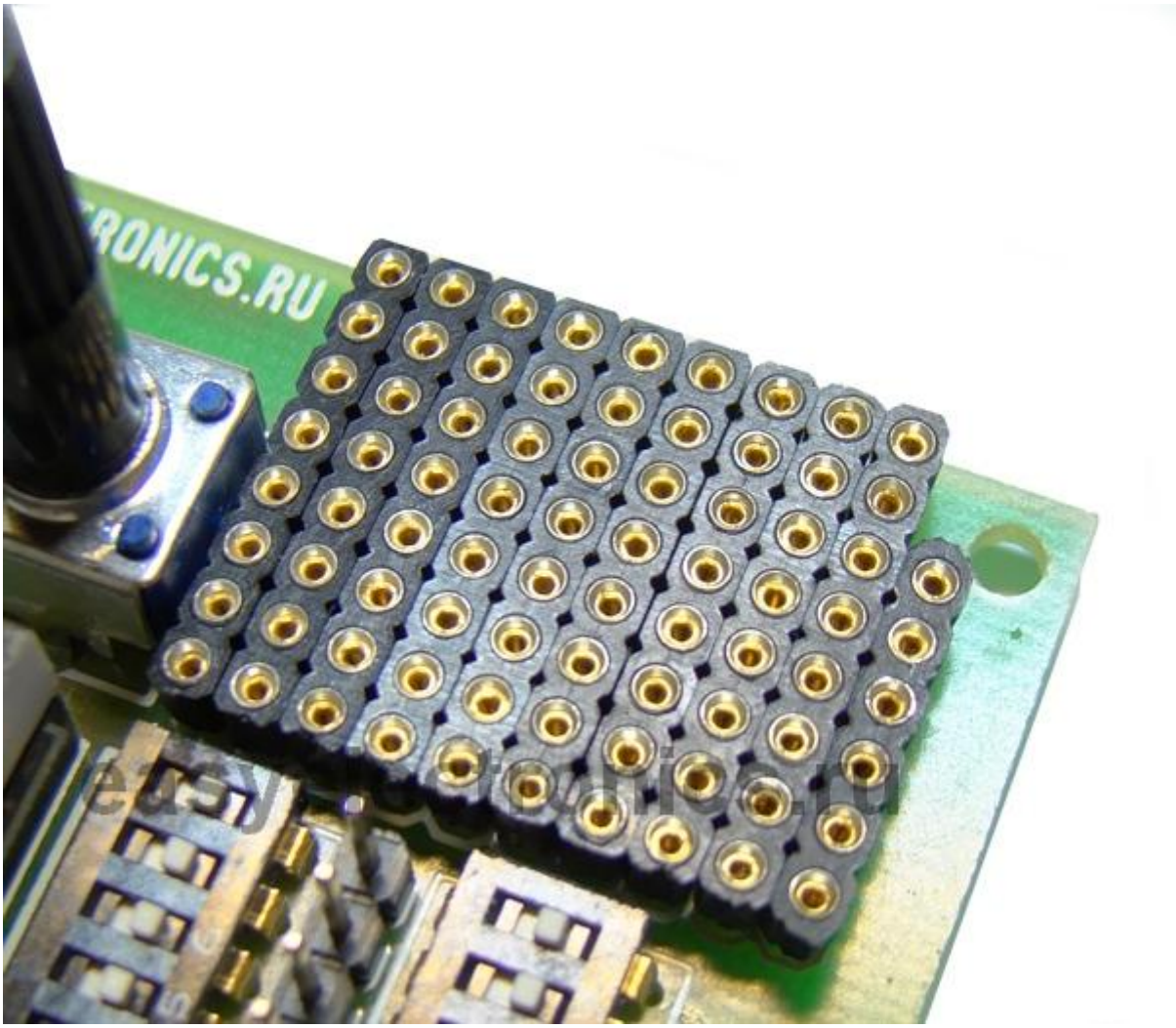
Также есть независимая RC цепочка из конденсатора и переменного резистора, что позволяет менять постоянную времени этой цепи. С помощью джампера эту RC цепь можно подключить к одному из входов АЦП и она послужит фильтром низких частот.



Три подстроечных многооборотных резисторов на 10к с выводами на штырях помогут организовать нужное сопротивление или послужат делителем напряжения.

Два переменных резистора, один из которых сразу подключен как делитель и может быть заведен на АЦП установкой джампера, другой же независимый и используется произвольно.

Небольшая цанговая макетная панель;
Если же возможностей платы вам вдруг не хватит и захочется собрать еще какой-либо узел, то к вашим услугам монтажная панель, с классической группировкой контактов.



Контакты цанговые, что обеспечивает надежную работу, а также отпадает нужда в специальных проводках. Цанговое гнездо плотно хватается даже за зачищенный проводок от витой пары, обеспечивая отличный контакт.

Преобразователь логических уровней
Простейший восьмиканальный резистивный делитель, неплохо зарекомендовавший

себя при связи мобильного телефона Siemens напомню он работает от 3.3 вольт) с 5ти вольтовым микроконтроллером.

Низкая цена
Гораздо ниже чем отладочные платы других производителей, при этом с куда более низким функционалом. Подробности и контакты в конце статьи.

Доставка
По России осуществляется посредством почты России. Стоимость доставки составляет 200р. Средний срок доставки полторы-две недели.

Также осуществляем доставку по ближнему и дальнему зарубежью, лишь бы не было ограничений на ввоз приборов.
Доставка в этом случае стоит 15$, отправляется обычной наземной почтой. Могу отправить и каким-нибудь DHL, но стоить это будет едва ли не столько же сколько вся плата :(— цены у них варварские.

Контроль качества
Все платы проходят предполетную цепную проверку работы и контроля качества.

Цепная проверка состоит из нескольких этапов.

- Визуальный осмотр. Проверяется красота и качество пайки. Наличие перемычек и непропая. Чтобы все детали стояли ровно и были впаяны правильно. Чтобы радовало глаз. Джамшутинг отправляется на переделку.
- Подключение к компьютеру. При этом происходит проверка работы преобразователя FTDI USB-USART
- Самопрошивка встроенным программатором управляющей программы JTAG'а. При этом проверяется работа программатора FTDI.
- Прошивка через интерфейс JTAG главного контроллера платы. При этом проверяется корректность работы интерфейса JTAG.
- Заливка демопрошивки через Bootloader. При этом проверяется работа бутлоадера.
- Проверка работы демопрошивки. Что дает визуальное подтверждение работы индикации платы.

- Проверка питания и контрольных напряжений.

В результате такой проверки происходит естественное и неизбежное тестирование всех ключевых узлов. Так что неисправная плата просто не покинет стол монтажника.

Комплектация:

- Собраная и протестированная плата с контроллером ATmega16. Контроллер прошит загрузчиком.
- LCD дисплей WH1602 текстовый, две строки по 16 символов. С подсветкой
- 2 кварца на 16 и на 12мгц.
- 1 метр разноцветного шлейфа
- 100 контактов BLS для изготовления соединительных проводков
- Кусок пупырчатого полиэтилена для достижения состояния душевного спокойствия.
- Полная документация высылается после покупки по электронной почте

Краткая Документация

- Краткое техническое руководство [1]
- Инструкции по быстрому старту [2]
- Инструкция по самопрошивке и программатору FTBB [3]
- Уроки по микроконтроллерам AVR [4]

Форум поддержки и контактная информация
Форма заказа [5]
forum.easyelectronics.ru — Pinboard [6]
Наличие, поставки, текущее состояние дел [7]

Гарантии
Если же так получилось, что вы, не смотря ни на что, получили неисправную плату, то я без разговоров обменяю ее на исправную, компенсируя почтовые расходы за свой счет.

Почта России? А дойдет?

~~Сто квитанций об отправке плат. Сто подтверждений о доставке~~ На данный момент уже около двухсот. Предоставляется треккинг по которому путь посылки можно отслеживать. Для сомневающихся — мы можем послать и почтовой службой EMS/DHL. Дотащат до двери, но, естественно, будет стоить дороже.

Да, в дальнейшем, все примеры Учебного Курса AVR [4] на моем сайте будут построены (а старые уроки подправлены и дополнены) на базе этой макетной платы. Имея на руках одинаковый инструментарий мне будет проще обьяснить что надо сделать, а вам будет проще повторить то, о чем я пишу — процесс обучения будет намного более эффективным.

Также планируются модули расширения, развивающие функционал демоплаты, добавляющий новые функции, превращающие ее в мощный микроконтроллерный конструктор

Как купить? В моем интернет магазине! Или пишите на shop@easyelectronics.ru, там же вы сможете задать свои вопросы по плате, доставке или способах оплаты [5]

AVR. Учебный курс. Работа с портами ввода-вывода. Практика

Вот ты читаешь сейчас это и думаешь — память, регистры, стек и прочее это хорошо. Но ведь это не пощупать, не увидеть. Разве что в симуляторе, но я и на дельфи с тем же условием могу накодить. Где мясо!!!
В других курсах там, чуть ли не с первых строк, делают что то существенное — диодиком мигают и говорят, что это наш Hello World. А тут? Гыде???

Да-да-да, я тебя понимаю. Более того, наверняка ты уже сбегал к конкурентам и помигал у них диодиком ;))))
Ничего, простительно.

Я просто не хотел на этом же мигании дидодиков и остановиться, а для прогресса нужно четкое понимание основ и принципов — мощная теоретическая база. Но вот пришла очередь практики.

О портах было рассказано, шаблон программы у вас уже есть, так что сразу и начнем.

Инструментарий
Работа с портами, обычно, подразумевает работу с битами. Это поставить бит, сбросить бит, инвертировать бит. Да, конечно, в ассемблере есть удобные команды

cbi/sbi, но работают они исключительно в малом адресном диапазоне (от 0 до 1F, поэтому давайте сначала напишем универсальные макросы, чтобы в будущем применять их и не парить мозг насчет адресного пространства.

Макросы будут зваться:

- SETB byte,bit,temp
- CLRB byte,bit,temp
- INVB byte,bit,temp,temp2

Причем при работе с битами младших PBB (0-1F адрес) то значение параметра TEMP можно и не указывать — он все равно подставляться не будет. За исключением команд инверсии — там промежуточные регистры полюбому нужны будут.

Также полезно заиметь группу макросов не использующих регистры. Точнее регистры они использовать будут, но предварительно сохранив их в стеке. Их можно будет бездумно пихать словно обычные команды. Но выполняться они будут дольше и будут требовать оперативной памяти.

- SETBM byte,bit
- CLRBM byte,bit
- INVBM byte,bit

Вот их исходный код. Как можно заметить, активно используются условия макроязыка, что дает возможность налупить универсальных макросов. Компилятор сам разберется какую версию куда ему подсунуть :)

```
1   ;= Start macro.inc ======================================
2   ;SET BIT with stack
3           .MACRO  SETBM
4           .if @0 < 0x20
5           SBI     @0,@1
6           .else
7                   .if @0<0x40
8           PUSH    R17
9           IN      R17,@0
10          ORI     R17,1<<@1
11          OUT     @0,R17
12          POP     R17
13                  .else
14          PUSH    R17
15          LDS     R17,@0
16          ORI     R17,1<<@1
17          STS     @0,R17
18          POP     R17
19                  .endif
20          .endif
21          .ENDM
22
23  ;SET BIT with REG
24          .MACRO  SETB
25          .if @0 < 0x20                   ; Low IO
26          SBI     @0,@1
27          .else
28                  .if @0<0x40             ; High IO
29          IN      @2,@0
30          ORI     @2,1<<@1
31          OUT     @0,@2
```

```
32                  .else                   ; Memory
33          LDS     @2,@0
34          ORI     @2,1<<@1
35          STS     @0,@2
36                  .endif
37          .endif
38          .ENDM
39  ;.....................................................
40  ;Clear BIT with REG
41          .MACRO  CLRB
42          .if @0 < 0x20                   ; Low IO
43          CBI     @0,@1
44          .else
45                  .if @0<0x40             ; High IO
46          IN      @2,@0
47          ANDI    @2,~(1<<@1)
48          OUT     @0,@2
49                  .else                   ; Memory
50          LDS     @2,@0
51          ANDI    @2,~(1<<@1)
52          STS     @0,@2
53                  .endif
54          .endif
55          .ENDM
56
57  ;Clear BIT with STACK
58          .MACRO  CLRBM
59          .if @0 < 0x20
60          CBI     @0,@1
61          .else
62                  .if @0<0x40
63          PUSH    R17
64          IN      R17,@0
65          ANDI    R17,~(1<<@1)
66          OUT     @0,R17
67          POP     R17
68                  .else
69          PUSH    R17
70          LDS     R17,@0
71          ANDI    R17,~(1<<@1)
72          STS     @0,R17
73          POP     R17
74                  .endif
75          .endif
76          .ENDM
77  ;.....................................................
78
79          .MACRO  INVB
80          .if     @0 < 0x40
81          IN      @2,@0
82          LDI     @3,1<<@1
83          EOR     @3,@2
84          OUT     @0,@3
85          .else
86          LDS     @2,@0
87          LDI     @3,1<<@1
88          EOR     @2,@3
89          STS     @0,@2
90          .endif
91          .ENDM
92
93          .MACRO  INVBM
94          .if     @0 < 0x40
```

```
95          PUSH    R16
96          PUSH    R17
97          IN      R16,@0
98          LDI     R17,1<<@1
99          EOR     R17,R16
100         OUT     @0,R17
101         POP     R17
102         POP     R16
103         .else
104         PUSH    R16
105         PUSH    R17
106         LDS     R16,@0
107         LDI     R17,1<<@1
108         EOR     R17,R16
109         STS     @0,R17
110         POP     R17
111         POP     R16
112         .endif
113         .ENDM
114
115 ;= End macro.inc ========================================
```

Со временем, когда пишешь на ассемблере, таких вот макросов становится очень и очень много. Они выносятся в отдельный файл и просто подключаются к любому твоему проекту, а написание кода становится легким и приятным.

Но вернемся к коду,
Мигнем уж светодиодиком то, наконец?

Да не вопрос. На демоплате уже смонтированы светодиоды, почему бы их не заюзать? Они висят на выводах порта PD4,PD5, PD7. Надо только одеть джамперы.

Сперва настроим эти порты на выход, для этого надо записать в регистр DDR единичку (вспоминаем статью про порты ввода вывода [1])

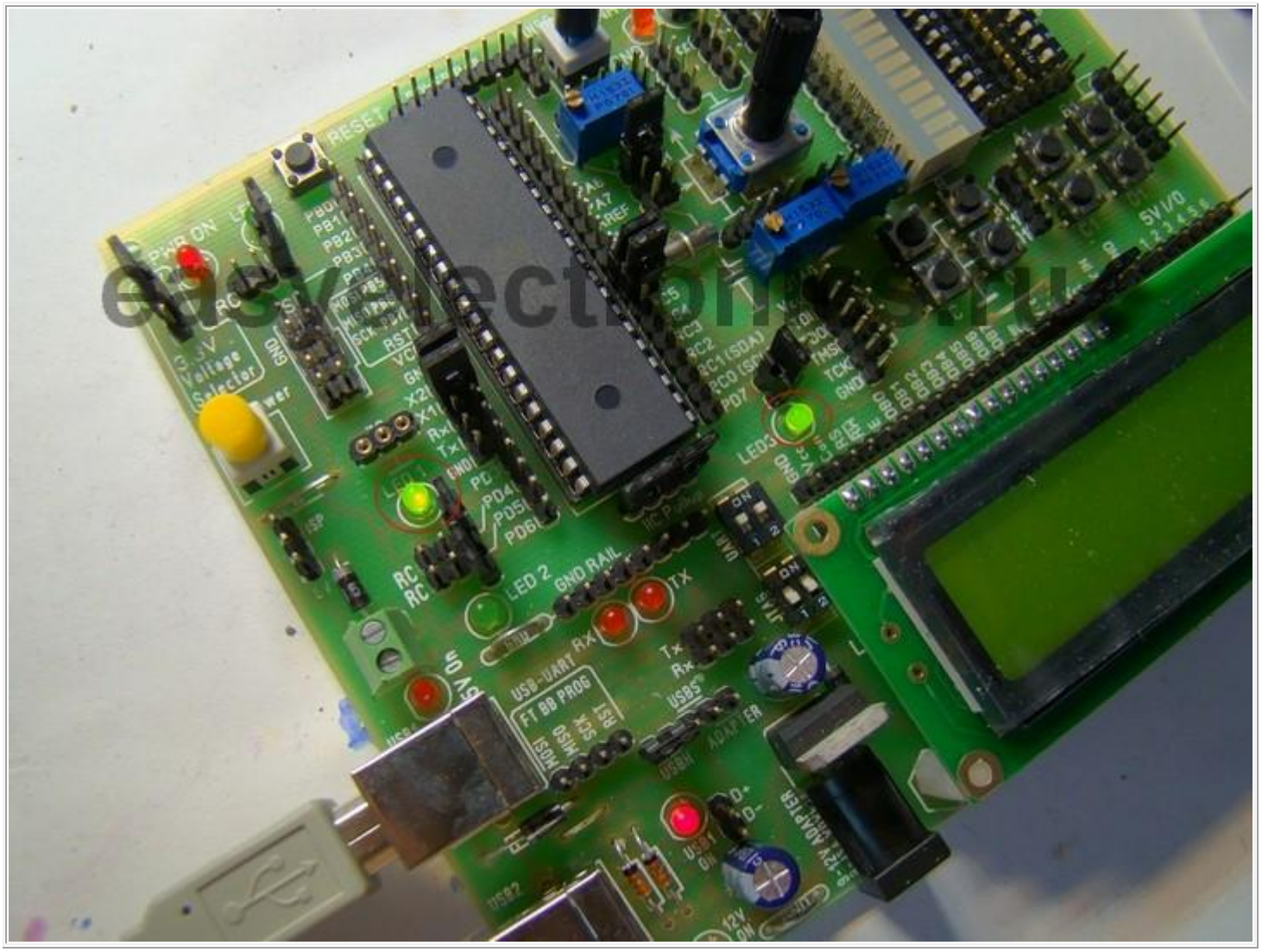
Сделать это можно где угодно по коду, но я все же рекомендую все настройки делать в секции инициализации нашего скелета.

```
1 ; Internal Hardware Init  =====================================
2
3         SETB    DDRD,4,R16      ; DDRD.4 = 1
4         SETB    DDRD,5,R16      ; DDRD.5 = 1
5         SETB    DDRD,7,R16      ; DDRD.7 = 1
6
7 ; End Internal Hardware Init ==================================
```

Осталось зажечь наши диоды. Зажигаются они записью битов в регистр PORT. Это уже делаем в главной секции программы.

```
1 ; Main ====================================================
2 Main:   SETB    PORTD,4,R16     ; Зажгли LED1
3         SETB    PORTD,7,R16     ; Зажгли LED3
4         JMP     Main
5 ; End Main ================================================
```

Компилиуем, можешь в трассировщике прогнать, сразу увидишь как меняются биты. Прошиваем… и после нажатия на RESET и выгрузки bootloader (если конечно у тебя Pinboard [2]) увидишь такую картину:



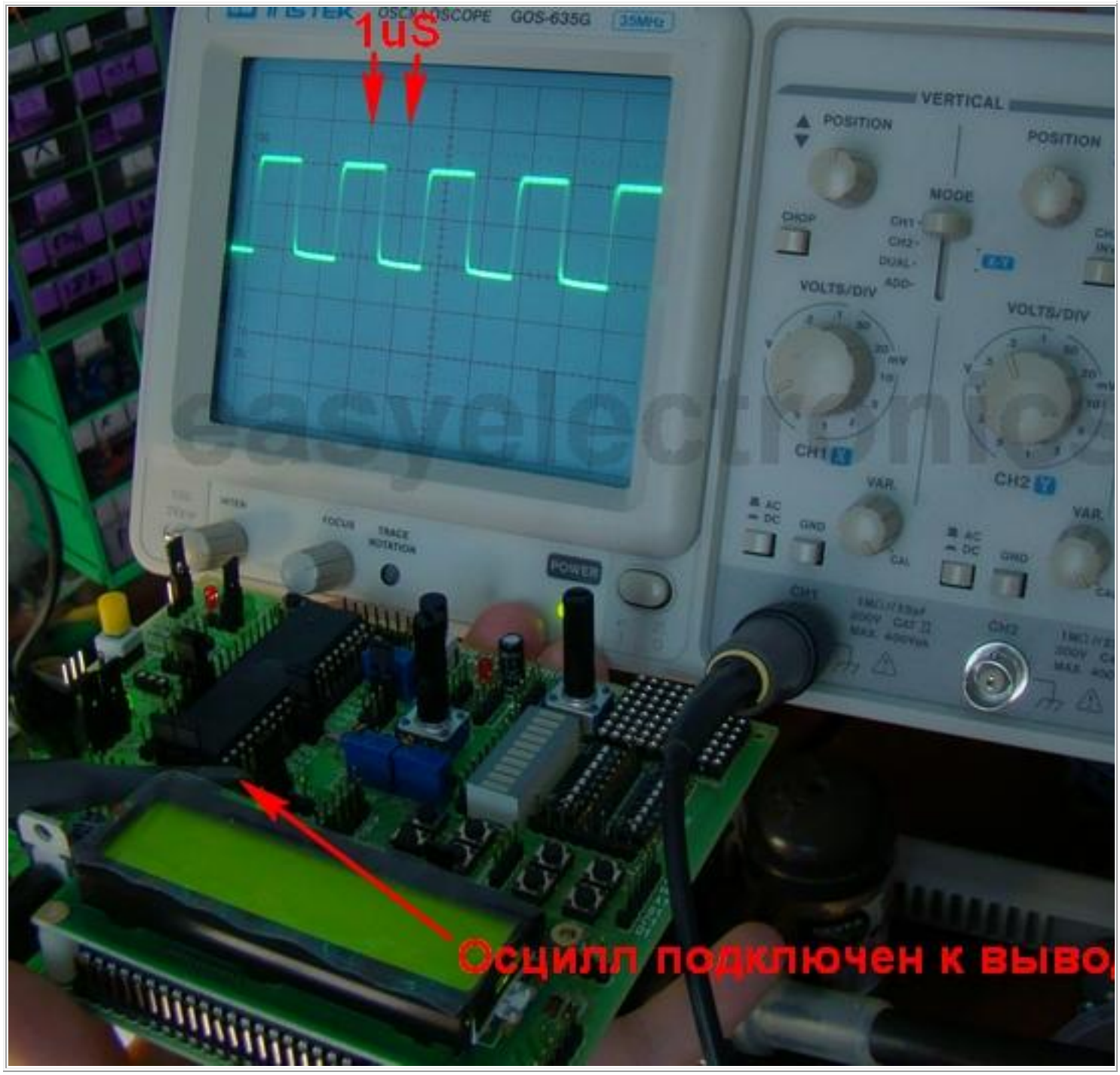
Во! Тока это же скучно. Давай ка ими помигаем.

Заменим всего лишь наши макрокоманды.

```
1 ; Main ====================================================
2 Main:   SETB    PORTD,4,R16             ; Зажгли LED1
3         INVB    PORTD,7,R16,R17         ; Инвертировали LED3
4         JMP     Main
5 ; End Main ================================================
```

Зажгли, прошили…

А вот фиг — горят оба, но один чуть-чуть тусклей. На самом деле он мерцает, но очень очень быстро. Если ткнуть осциллографом в вывод PD7, то будет видно, что уровень меняется там с бешеной частотой:



Что делать? Очевидно замедлить. Как? Самый простой способ, который практикуют в подавляющем большинстве обучалок и быстрых стартов — тупой задержкой. Т.е. получают код вида:

```
1  ; Main ================================================
2  Main:   SETB    PORTD,4,R16             ; Зажгли LED1
3          INVB    PORTD,7,R16,R17         ; Инвертировали LED3
4
5          RCALL   Delay
6
7          JMP     Main
8  ; End Main ============================================
9
```

```
10 ; Procedure ==================================================
11
12 .equ    LowByte  = 255
13 .equ    MedByte  = 255
14 .equ    HighByte = 255
15
16 Delay:  LDI     R16,LowByte     ; Грузим три байта
17        LDI     R17,MedByte     ; Нашей выдержки
18        LDI     R18,HighByte
19
20 loop:   SUBI    R16,1           ; Вычитаем 1
21        SBCI    R17,0           ; Вычитаем только C
22        SBCI    R18,0           ; Вычитаем только C
23
24        BRCC    Loop            ; Если нет переноса - переход
25        RET
26 ; End Procedure ==============================================
```

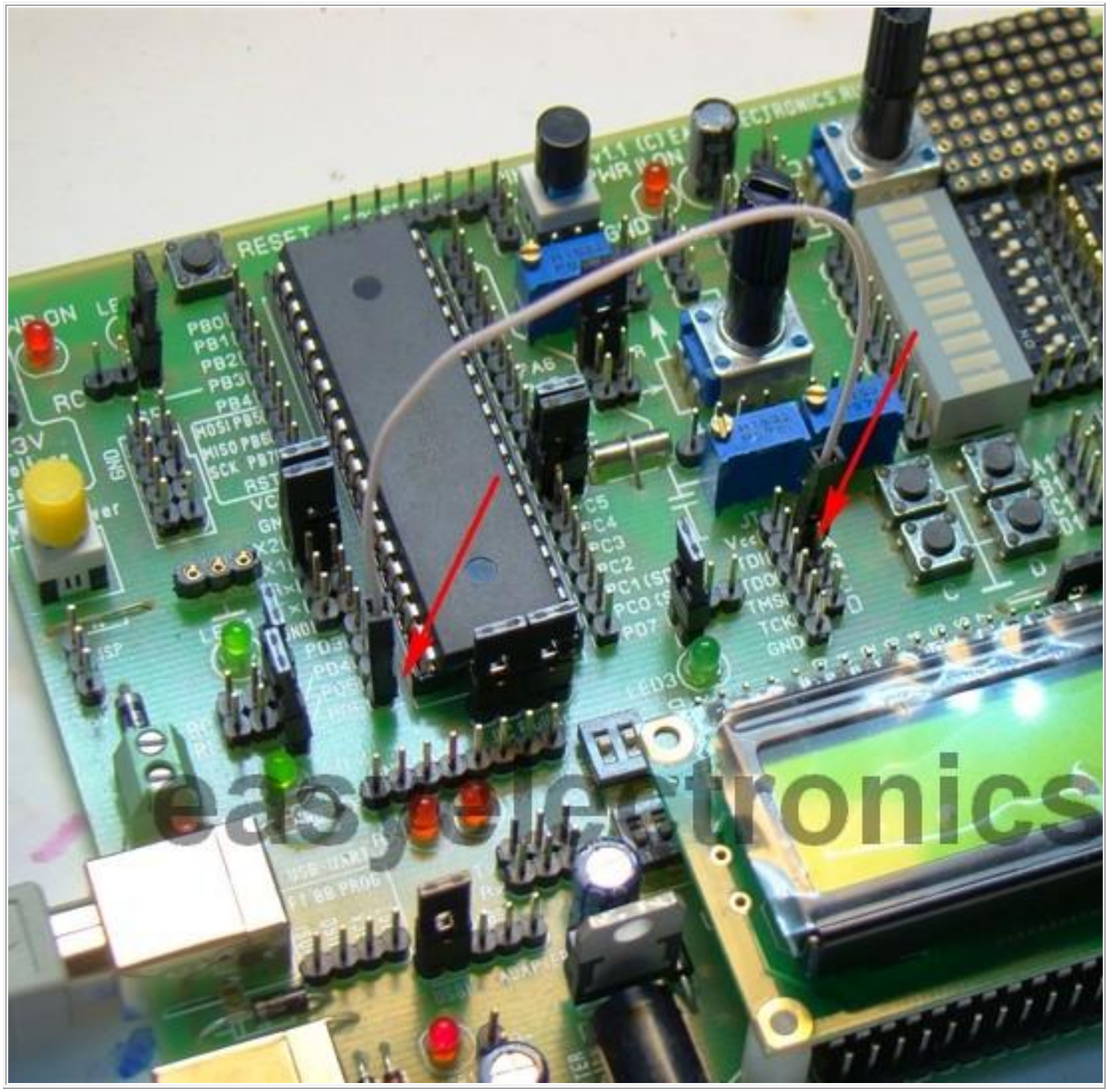
Прошили, запустили… О да, теперь мигание будет заметно.

При параметрах 255.255.255 длительность выдержки на 8Мгц будет около 2.1 секунды. Можно увеличить разрядность задержки еще на несколько байт. Тогда можно хоть час зарядить.

Но этот метод ущербен, сейчас покажу почему.

Давай добавим кнопку. LED3 пусть мигает в инверсии. А мы сделаем так, что когда кнопка нажата у нас горит LED1, а когда отпущена горит LED2.

В качестве кнопки возьмем тактовую A, подключим ее к порту PD6.



Проверка кнопки делается командой SBIC, но вначале ее надо инициализировать. Сделать DDR=0, PORT=1 — вход с подтяжкой.

Добавляем в секцию инициализации (Internal Hardware Init) эти строчки:

```
1         SETB    PORTD,6,R16
2         CLRB    DDRD,6,R16
```

Усе, все настроено как надо :)

Осталось теперь проверить нажата ли кнопка или нет. Если нажата, то в бите PIND.6 будет 0.

```
1  ; Main =====================================================
```

```
2  Main:          SBIS    PIND,6                ; Если кнопка нажата - переход
3                 RJMP    BT_Push
4
5                 SETB    PORTD,5               ; Зажгем LED2
6                 CLRB    PORTD,4               ; Погасим LED1
7
8  Next:          INVB    PORTD,7,R16,R17       ; Инвертировали LED3
9
10                RCALL   Delay
11
12                JMP     Main
13
14
15 BT_Push:       SETB    PORTD,4               ; Зажгем LED1
16                CLRB    PORTD,5               ; Погасим LED2
17
18                RJMP    Next
19
20 ; End Main ======================================================
```

Ну чо, работает. Кнопочка жмется — диодики меняются. Третий же бодро подмигивает. Но есть западло:

Торомозит программка то! Я кнопочку нажал, а картинка не сменилась, нужно подождать, подержать… Почему? А это из-за нашего быдлокодинга.

Помнишь я тебе говорил, что тупые задержки, в которых МК ничего не делает это адское зло? Вот! Теперь ты в этом убедился сам. Чтож, со злом надо бороться. Как? Ну эт я тоже уже говорил — делать непрерывный цикл с флажками. Хотя бы внести в нашу задержку полезную работу.

Шарманка
Сейчас я тебе покажу как можно сделать цифровую шарманку. Помнишь как она устроена?

Там барабан с торчащими гвоздями и пружинки на разные тона. Гвозди вращаются, дергают пружинки — они звякают. Получается расколбасный музон. А что если нашу шарманку развернуть в ленту. Не правда ли гвозди похожи на единички ? ;))))

Лентой будет счетчик, считающий от нуля, скажем, до FF.FF.FF.FF и потом опять до нуля или еще какой величины, сколько надо столько и сделаем. А наши подпрограммы будут играть роль пружинок, цепляясь в нужных местах — сравнивая свою константу с текущим значением счетчика.

Совпало? Делаем «ДРЫНЬ!»

Осталось только прописать на временном цикле нашей шарманки где и что должно запускаться. И тут есть одна очень удобная особенность — для построения циклических последовательностей нам достаточно отлавливать один разряд.

Скажем, считает шарманка от 0 до 1000, а нам надо 10 раз мигнуть диодом. Не обязательно втыкать 10 обработчиков с разными значениями. Достаточно одного, но чтобы он ловил значение **10. Все, остальное нам не важно. И сработает он на 0010, 0110, 0210, 0310, 0410, 0510, 0610, 0710, 0810, 0910. Более частые интервалы также делятся как нам надо, достаточно влезть в другой разряд. Тут надо только не забыть отрезать нафиг старшие разряды, чтобы не мешались.

А учитывая что считать мы там будем в двоичном системе, то ловить кодовую посылку станет еще приятней — хлоп ее по битмаске и все. Даже проверять все байты не нужно.

Приступим. Вначале создадим наш счетчик в сегменте данных:

```
1 ; RAM =============================================================
2                 .DSEG
3 CCNT:   .byte   4
```

Дальше сформируем цикл счетчика, нам надо чтобы он тикал каждый оборот. Данные лежат в памяти, поэтому для тиканья придется их загрузить в регистры, а потом выгрузить обратно.

```
1                LDS     R16,CCNT
2                LDS     R17,CCNT+1
3                LDS     R18,CCNT+2
4                LDS     R19,CCNT+3
```

Все, теперь в R16 самый младший байт нашего счетчика, а в R19 самый старший.

Регистры можно предварительно затолкать в стек, но я дам тебе лучше другой совет — когда пишешь программу, продумывай алгоритм так, чтобы использовать регистры как сплошной TEMP данные которого актуальны только здесь и сейчас. А что будет с ними в следующей процедуре уже не важно — все что нужно должно будет сохранено в оперативке.

Дальше надо к этому четырехбайтному числу прибавить 1. Да так, чтобы учесть все переполнения.

По можно сделать так:

```
1                LDI     R20,1           ; Нам нужна единичка
2                CLR     R15             ; А еще нолик.
3
4                ADD     R16,R20 ; Прибавляем 1 если в регистре 255, то будет C
5                ADC     R17,R15 ; Прибавляем 0+C
6                ADC     R18,R15 ; Прибавляем 0+C
7                ADC     R19,R15 ; Прибавляем 0+C
```

Пришлось потратить еще два регистра на хранение констант нашего сложения. Все от того, что AVR не умеет складывать регистры с непосредственным числом. Зато умеет вычитать.

Я уже показывал, что R-(-1)=R+1, а ведь никто не запрещает нам этот же прием устроить и тут — сделать сложение через вычитание.

```
1                SUBI    R16,(-1)
2                SBCI    R17,(-1)
3                SBCI    R18,(-1)
4                SBCI    R19,(-1)
```

Даст нам инкремент четырехбайтного числа R19:R18:R17:R16

А теперь я вам покажу немного целочисленной магии
Почему это будет работать? Смотри сам:

SUBI R16,(-1) это, по факту R16 — 255 и почти при всех раскладах она нам даст нам заём из следующего разряда — C. А в регистре останется то число на которое больше.

Т.е. смотри как работает эта математика, вспомним про число в доп кодах. Покажу на четырехрязрядном десятичном примере. У Нас есть ВСЕГО ЧЕТЫРЕ РАЗРЯДА, ни больше ни меньше. Отрицательное число это 0-1 так? Окей.

1_C 0000-1=9999+C

Т.е. мы как бы взяли как бы из пятиразрядной 1_C 0000 отняли 1, но разрядов то у нас всего четыре! Получили доп код 9999 и флаг заема C (сигнализирующий о том, что был заем)

Т.е. в нашей целочисленной математике 9999=-1 :) Проверить легко -1+1 = 0 Верно?

9999+1 = 1_C 0000 Верно! :))) А 1 старшего разряда банально не влезла в разрядность и ушла в флаг переноса C, сигнализирующего еще и о переполнении.

Оки, а теперь возьмем и сделаем R-(-1). Пусть R=4

1_C0004-9999 = 0005+C

Вот так вот взяли и сложили через вычитание. Просто магия, да? ;)

Прикол ассемблера в том, что это всего лишь команды, а не доктрина и не правило. И команды которые подразумевают знаковые вычисления можно использовать где угодно, лишь бы они давали нужный нам результат!

Вот и тут — наш счетчик он же беззнаковый, но мы используем особенности знакового исчисления потому что нам так удобней .

Флаг C не выскочит лишь когда у нас дотикает до 255 (9999 в десятичном примере), тогда будет 255-255 = 0 и вскочит лишь Z, но нам он не нужен.

А дальше, вторая команда работает также, только с учетом знака

```
1                SBCI    R17,(-1)
```

Т.е. R17-(-1)-1_C, а поскольку C у нас всегда (кроме последнего случая перед переполнением), то до тех пор пока значение R16 не достигнет 255 (и флаг C не пропадет) у нас будет R17+1-1_C=R17. Как только флаг пропадет, тогда и случится инкремент второго байта R17+1-0_C. А там по цепочке и все остальные.

Короче, хоть 100 байтную переменную создавай и таким образом ее прощелкивай.

А потом выгружаешь регистры обратно в память:

```
1                STS     CCNT,R16
2                STS     CCNT+1,R17
3                STS     CCNT+2,R18
4                STS     CCNT+3,R19
```

Код Инкремента четырехбайтной константы в памяти можно свернуть в макрос, чтобы не загромождать код

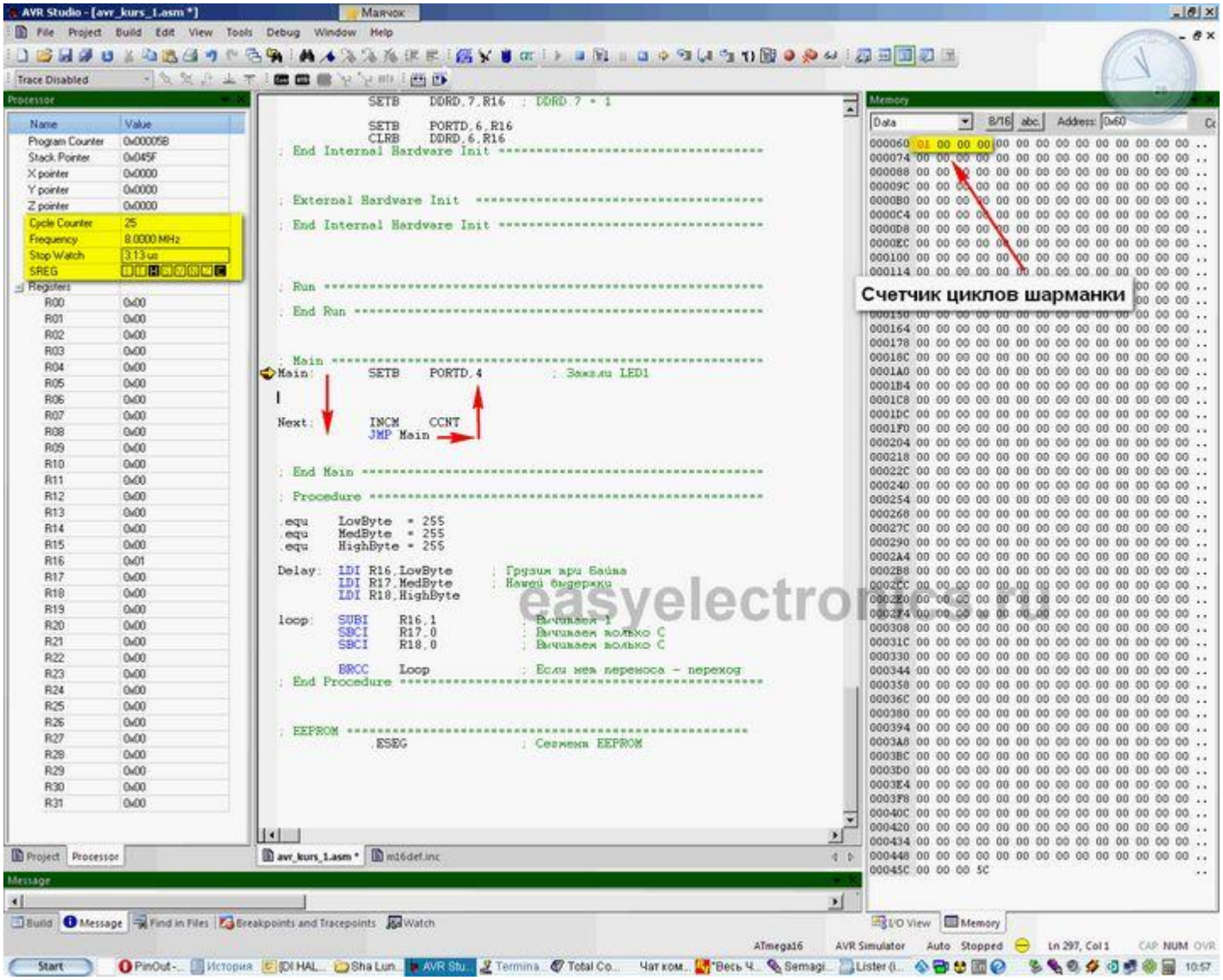
```
1                 .MACRO  INCM
2                 LDS     R16,@0
3                 LDS     R17,@0+1
4                 LDS     R18,@0+2
5                 LDS     R19,@0+3
6
7                 SUBI    R16,(-1)
8                 SBCI    R17,(-1)
9                 SBCI    R18,(-1)
10                SBCI    R19,(-1)
11
12                STS     @0,R16
13                STS     @0+1,R17
14                STS     @0+2,R18
15                STS     @0+3,R19
16                .ENDM
```

Теперь осталось прикинуть как будет мигать наш диодик по шарманке. Для этого давай посчитаем длительность цикла.

```
1 ; Main ============================================================
2
3 Main:   SETB    PORTD,4         ; Зажгли LED1
4
5
```

```
6           INVB    PORTD,7,R16,R17; Инвертировали LED3
7
8 Next:     INCM    CCNT
9           JMP     Main
```

Запусти режим отладки и поставь точку останова (F9) на метку Main и загони курсор на первый брейкпоинт.



Увеличить [3]

Сбрось StopWatch и Cycle Counter, а потом сделай одну итерацию цикла. У меня студия показала 25 машинных циклов и 3.3 микросекунды времени. Это длительность одной итерации.

Теперь посчитаем сколько нам надо таких итераций на секунду. 1/0.75E-6 = 303 030 оборота. Что дофига :) Посчитаем это в байтах нашей шарманки, просто переведем в 16тиричную систему. Получим 0x049FB6 или, если побайтно, то

```
1 0xB6(CCNT)
2 0x9F(CCNT+1)
3 0x04(CCNT+2)
4 0x00(CCNT+3)
```

Осталось теперь только сравнить число с этим слепком.

Как сравнивать? Да довольно просто. Все зависит от того что мы хотим получить. Если ОДНО событие за ВЕСЬ период глобального счетчика нашей шарманки, то тупо, побайтно. В этом случае у тебя диодик моргнет через секунду после старта, а дальше ты будешь ждать пол часа до переполнения всего четырехбайтного счетчика.

Чтобы он моргал каждую секунду тебе надо при сравнении замаскировать старшие биты глобального счетчика, словно у него разрядность не 32 бита, а меньше (и переполняется он чаще).

Младшие байты сравниваем как есть, а самый старший только до его максимального разряда, остальные надо отрезать.

Т.е. самый старший разряд для этого случая это CCNT+2=0x04 если в двоичном представлении то 0x04 = **00000**100 так вот, счетчик у нас четырех разрядный, значит событие с маской

00 04 9F B6 (**00000000 00000**100 10011111 10110110)

до переполнения возникнет дофига число раз. Видишь я нули жирным шрифтом выделил. Старший самый у мы вообще сравнивать не будем, а вот пред старший надо через AND по маске 00000111 продавить, чтобы отсечь старшие биты.

Их еще надо заполнить до переполнения и обнуления счетчика. Но если мы их замаскируем то их дальнейшая судьба нас не волнует. Пусть хоть до второго пришествия тикает — нам плевать.

```
1                LDS     R16,CCNT        ; Грузим числа в регистры
2                LDS     R17,CCNT+1
3                LDS     R18,CCNT+2
4                ANDI    R18,0x07        ; Накладываем маску
5
6                CPI     R16,0xB6        ; Сравниванем побайтно
7                BRNE    NoMatch
8                CPI     R17,0x9F
9                BRNE    NoMatch
10               CPI     R18,0x04
11               BRNE    NoMatch
12
13 ; Если совпало то делаем экшн
14 Match:        INVB    PORTD,7,R16,R17 ; Инвертировали LED3
15
16 ; Не совпало - не делаем :)
17 NoMatch:
18
19 Next:         INCM    CCNT            ; Проворачиваем шарманку
20               JMP     Main
```

Во, загрузили теперь мигает. Никаких затупов нет, нигде ничего не подвисает, а главный цикл пролетает со свистом, только успевай барабан шарманки проворачивать :)

Вот только мигает заметно медленней чем мы хотели. Не 1 секунда, а 8. Ну а что ты хотел — добавив процедуру сравнения мы удлиннили цикл еще на несколько команд. И теперь он выполняется не 25 тактов, а 36. Пересчитывай все циферки заново :)))))

Но это еще не самый цимес! Прикол в том, что у тебя часть кода выполняется, а часть нет — команды сравнения и перехода. Поэтому точно высчитать задержку по тактам это проще сразу удавиться — надо по каждой итерации высчитать когда и сколько у тебя будет переходов, сколько они тактов займут…

А если код будет еще больше, то ваще труба и погрешность накапливается с каждой итерацией!

Зато, если добавить код обработки кнопок:

```
1  ; Main ================================================
2  Main:          SBIS    PIND,6          ; Если кнопка нажата - переход
3                 RJMP    BT_Push
```

```
4
5                   SETB    PORTD,5 ; Зажгем LED2
6                   CLRB    PORTD,4 ; Погасим LED1
7
8  Next:            LDS     R16,CCNT        ; Грузим числа в регистры
9                   LDS     R17,CCNT+1
10                  LDS     R18,CCNT+2
11                  ANDI    R18,0x07
12
13                  CPI     R16,0xB6        ; Сравниванем побайтно
14                  BRNE    NoMatch
15                  CPI     R17,0x9F
16                  BRNE    NoMatch
17                  CPI     R18,0x04
18                  BRNE    NoMatch
19
20 ; Если совпало то делаем экшн
21 Match:           INVB    PORTD,7,R16,R17 ; Инвертировали LED3
22
23 ; Не совпало - не делаем :)
24 NoMatch:         NOP
25
26
27                  INCM    CCNT
28                  JMP     Main
29
30
31 BT_Push:         SETB    PORTD,4 ; Зажгем LED1
32                  CLRB    PORTD,5 ; Погасим LED2
33                  RJMP    Next
34
35 ; End Main =================================================
```

То увидим, что от тормозов и следов не осталось. Кнопки моментально реагируют на нажатия, а диодик мигает сам по себе. Многозадачность! :)

В общем, шарманка не годится там где нужны точные вычисления. Но если задача стоит из серии «надо периодически дрыгать и не принципиально как точно», то самое то. Т.к. не занимает аппаратных ресурсов вроде таймера.

Например, периодически сканировать клавиатуру, скажем, каждые 2048 оборотов главного цикла. Сам прикинь какое число надо нагрузить на сравнение и какую маску наложить :)

Можешь скачать и поглядеть **проект с шарманкой** [4], протрассировать его, чтобы увидеть как там все вертится.

А для точных вычислений времени существуют таймеры. Но о них разговор отдельный.

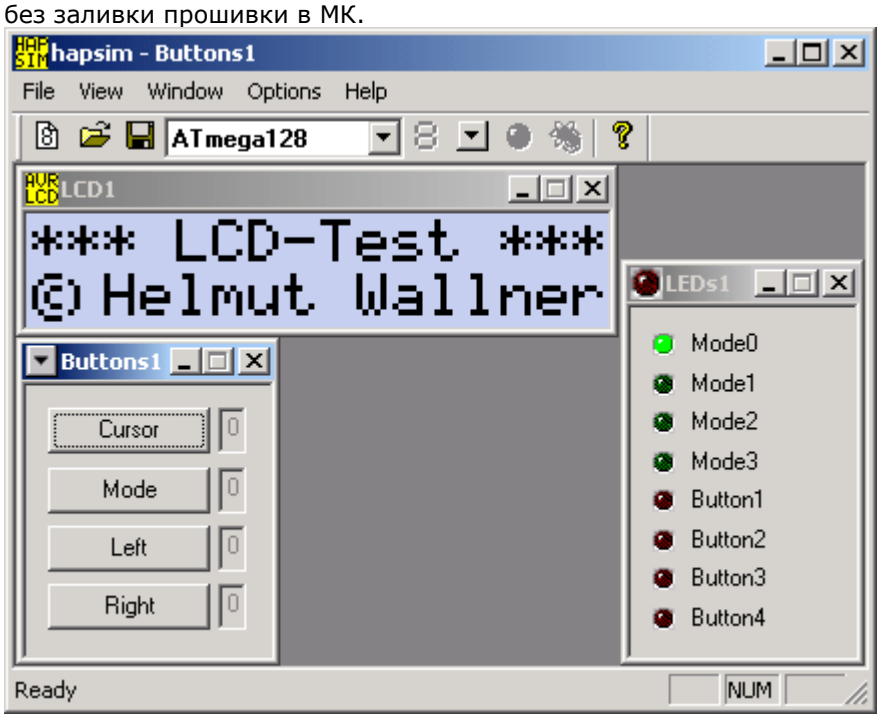
AVR Studio и HAPSim

Раз уж пошел разговор об отладке **AVR Studio**, то стоит упомянуть про ряд примочек для нее. Одной из самых интересных примочек является небольшая программка — **HAPSim**. Программка почти не известна, по крайней мере я лишь пару раз слышал упоминание о ней на форуме. Прога не слишком стабильна, иногда работает не так как надо, а еще ей не хватает многих эргономических моментов, но, надеюсь, в следующий версиях это будет исправлено.

Итак, что делает **Hapsim**. Он всего лишь подцепляется к студии и в момент выполнения эмуляции программы может симулировать работу разных устройств, прицепленных на порты. Из доступных устройств есть:

- Кнопки. Как обычные, прижимающие к земле, так и трехвыводные, которые могут поставить как высокий, так и низкий уровень в зависимости от состояния
- LCD на базе HD44780
- Терминал (USART или I^2C) с возможностью осуществить трансфер на комповый COM порт (сам не проверял, но заявлено в мануале)
- Клавиатурная матрица 4x4
- Блок светодиодов

Не богато, но вполне хватает для простенькой отладки. Особо радует терминал :) Можно по быстрому отлаживать без заливки прошивки в МК.



Как работать с Hapsim?
А тут все гораздо проще чем кажется:

1. Распаковать Hapsim куда-нибудь рядом с студией. Я прям в ее каталог сунул, в папочку.
2. Запустить **AVR Studio** с нужным проектом.
3. Запустить программку.
4. Выбрать МК, не в пример протеусам всяким, **тут поддерживается вся линейка AVR**.
5. Добавить в окно Hapsim нужные тебе девайсы (File -> New Control), а затем настроить их. Каждый появившийся в рабочем поле девайс получает кнопку на панели инструментов. Тыкнул на нее — вот тебе и настройка. Если блоков несколько, то настройка открывается для активного окна. Настройки тоже не мудреные — указать бит порта, да активный режим. Например, для светодиодов можно указать цвет, название, порт и бит порта, а также галочкой сделать его инверсным (например, как я люблю, зажигать нулем)
6. Запустить в студии эмуляцию отладки и можно клацать по виртуальным кнопочкам и смотреть как ведет себя студия.

Замечу только, что в пошаговом режиме работает весьма стремно, особенно если надо нажать кнопку и удерживать (в хапсиме нет фиксируемых тумблеров), поэтому лучше запускать на исполнение (F5 — Run), а саму программу утыкать брейкпоинтами во всех ключевых местах.

На **сайте автора программы** [1] можно найти последнюю версию и примеры. Также Hapsim можно скачать у меня [2], весит он всего пол мегабайта.

AVR. Учебный курс. Таймеры

С счетчиком итераций главного цикла мы разобрались и выяснили, что для точных временных отсчетов он не годится совершенно — выдержка плавает, да и считать ее сложно. Что делать?

Очевидно, что нужен какой то внешний счетчик, который тикал бы независимо от работы процессора, а процессор мог в любой момент посмотреть что в нем такое натикало. Либо чтобы счетчик выдавал события по переполнению или опустошению — флажок поднимал или прерывание генерил. А проц это прочухает и обработает.

И такой счетчик есть, даже не один — это периферийные таймеры. В AVR их может быть несколько штук да еще с разной разрядностью. В ATmega16 три, в ATmega128 четыре. А в новых МК серии AVR может даже еще больше, не узнавал.

Причем таймер может быть не просто тупым счетчиком, таймер является одним из самых навороченных (в плане альтернативных функций) периферийных девайсов.

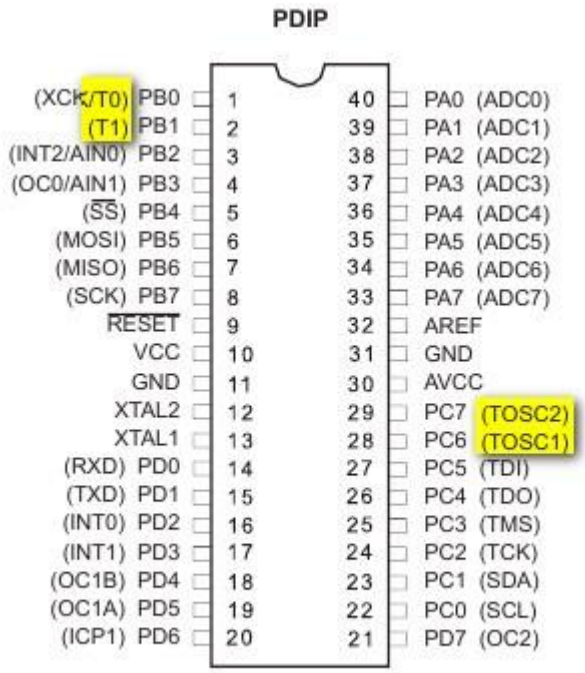
Что умееют таймеры

- Тикать с разной скоростью, подсчитывая время
- Считать входящие извне импульсы (режим счетчика)
- Тикать от внешнего кварца на 32768гц
- Генерировать несколько видов ШИМ сигнала
- Выдавать прерывания (по полудесятку разных событий) и устанавливать флаги

Разные таймеры имеют разную функциональность и разную разрядность. Это подробней смотреть в даташите.

Источник тиков таймера
Таймер/Счетчик (далее буду звать его T/C) считает либо тактовые импульсы от встроенного тактового генератора, либо со счетного входа.

Погляди внимательно на распиновку ног ATmega16, видишь там ножки T1 и T0?



Так вот это и есть счетные входы Timer 0 и Timer 1. При соответствующей настройке T/C будет считать либо передний (перепад с 0-1), либо задний (перепад 1-0) фронт импульсов, пришедших на эти входы.

Главное, чтобы частота входящих импульсов не превышала тактовую частоту процессора, иначе он не успеет обработать импульсы.

Кроме того, T/C2 способен работать в асинхронном режиме. То есть T/C считает не тактовые импульсы процессора, не входящие импульсы на ножки, а импульсы своего собственного собственного генератора, работающего от отдельного кварца. Для этого у T/C2 есть входы TOSC1 и TOSC2, на которые можно повесить кварцевый резонатор.

Зачем это вообще надо? Да хотя бы организовать часы реального времени. Повесил на них часовой кварц на 32768 Гц да считай время — за секунду произойдет 128 переполнений (т.к. T/C2 восьми разрядный). Так что одно переполнение это 1/128 секунды. Причем на время обработки прерывания по переполнению таймер не останавливается, он также продолжает считать. Так что часы сделать плевое дело!

Предделитель
Если таймер считает импульсы от тактового генератора, или от своего внутреннего, то их еще можно пропустить через предделитель.

То есть еще до попадания в счетный регистр частота импульсов будет делиться. Делить можно на 8, 32, 64, 128, 256, 1024. Так что если повесишь на T/C2 часовой кварц, да пропустишь через предделитель на 128, то таймер у тебя будет тикать со скоростью один тик в секунду.

Удобно! Также удобно юзать предделитель когда надо просто получить большой интервал, а единственный источник тиков это тактовый генератор процессора на 8Мгц, считать эти мегагерцы задолбаешься, а вот если пропустить через предделитель, на 1024 то все уже куда радужней.

Но тут есть одна особенность, дело в том, что если мы запустим T/C с каким нибудь зверским предделителем, например на 1024, то первый тик на счетный регистр придет не обязательно через 1024 импульса.

Это зависит от того в каком состоянии находился предделитель, а вдруг он к моменту нашего включения уже досчитал почти до 1024? Значит тик будет сразу же. Предделитель работает все время, вне зависимости от того включен таймер или нет.

Поэтому предделители можно и нужно сбрасывать. Также надо учитывать и то, что предделитель един для всех счетчиков, поэтому сбрасывая его надо учитывать то, что у другого таймера собьется выдержка до следующего тика, причем может сбиться конкретно так.

Например первый таймер работает на выводе 1:64, а второй на выводе 1:1024 предделителя. У второго почти дотикало в предделителе до 1024 и вот вот должен быть тик таймера, но тут ты взял и сбросил предделитель, чтобы запустить первый таймер точно с нуля. Что произойдет? Правильно, у второго делилка тут же скинется в 0 (предделитель то единый, регистр у него один) и второму таймеру придется ждать еще 1024 такта, чтобы получить таки вожделенный импульс!

А если ты будешь сбрасывать предделитель в цикле, во благо первого таймера, чаще чем раз в 1024 такта, то второй таймер так никогда и не тикнет, а ты будешь убиваться головой об стол, пытаясь понять чего это у тебя второй таймер не работает, хотя должен.

Для сброса предделителей достаточно записать бит PSR10 в регистре SFIOR. Бит PSR10 будет сброшен автоматически на следующем такте.

Счетный регистр
Весь результат мучений, описанных выше, накапливается в счетном регистре TCNTx, где вместо x номер таймера. он может быть как восьмиразрядным, так и шестнадцати разрядным, в таком случае он состоит из двух регистров TCNTxH и TCNTxL — старший и младший байты соответственно.

Причем тут есть подвох, если в восьмиразрядный регистр надо положить число, то нет проблем OUT TCNT0,Rx и никаких гвоздей, то с двухбайтными придется поизвращаться.

А дело все в чем — таймер считает независимо от процессора, поэтому мы можем положить вначале один байт, он начнет считаться, потом второй, и начнется пересчет уже с учетом второго байта.

Чувствуете лажу? Вот! Таймер точное устройство, поэтому грузить его счетные регистры надо одновременно! Но как? А инженеры из Atmel решили проблему просто:
Запись в старший регистр (TCNTxH) ведется вначале в регистр TEMP. Этот регистр чисто служебный, и нам никак недоступен.

Что в итоге получается: Записываем старший байт в регистр TEMP (для нас это один хрен TCNTxH), а затем записываем младший байт. В этот момент, в реальный TCNTxH, заносится ранее записанное нами значение. То есть два байта, старший и младший, записываются одновременно! Менять порядок нельзя! Только так

Выглядит это так:

```
1        CLI                    ; Запрещаем прерывания, в обязательном порядке!
2        OUT     TCNT1H,R16     ; Старшей байт записался вначале в TEMP
3        OUT     TCNT1L,R17     ; А теперь записалось и в старший и младший!
4        SEI                    ; Разрешаем прерывания
```

Зачем запрещать прерывания? Да чтобы после записи первого байта, прога случайно не умчалась не прерывание, а там кто нибудь наш таймер не изнасиловал. Тогда в его регистрах будет не то что мы послали тут (или в прерывании), а черти что. Вот и попробуй потом такую багу отловить! А ведь она может вылезти в самый неподходящий момент, да хрен поймаешь, ведь прерывание это почти случайная величина. Так что такие моменты надо просекать сразу же.

Читается все также, только в обратном порядке. Сначала младший байт (при этом старший пихается в TEMP), потом старший. Это гарантирует то, что мы считаем именно тот байт который был на данный момент в счетном регистре, а не тот который у нас натикал пока мы выковыривали его побайтно из счетного регистра.

Контрольные регистры
Всех функций таймеров я расписывать не буду, а то получится неподъемный трактат, лучше расскажу о основной — счетной, а всякие ШИМ и прочие генераторы будут в другой статье. Так что наберитесь терпения, ну или грызите даташит, тоже полезно.

Итак, главным регистром является TCCRx
Для T/C0 и T/C2 это TCCR0 и TCCR2 соответственно, а для T/C1 это TCCR1B

Нас пока интересуют только первые три бита этого регистра:
CSx2.. CSx0, вместо x подставляется номер таймера.
Они отвечают за установку предделителя и источник тактового сигнала.

У разных таймеров немного по разному, поэтому опишу биты CS02..CS00 только для таймера 0

- 000 — таймер остановлен
- 001 — предделитель равен 1, то есть выключен. таймер считает тактовые импульсы
- 010 — предделитель равен 8, тактовая частота делится на 8
- 011 — предделитель равен 64, тактовая частота делится на 64
- 100 — предделитель равен 256, тактовая частота делится на 256
- 101 — предделитель равен 1024, тактовая частота делится на 1024
- 110 — тактовые импульсы идут от ножки T0 на переходе с 1 на 0
- 111 — тактовые импульсы идут от ножки T0 на переходе с 0 на 1

Прерывания
У каждого аппаратного события есть прерывание, вот и таймер не исключение. Как только происходит переполнение или еще какое любопытное событие, так сразу же вылазит прерывание.

За прерывания от таймеров отвечают регистры TIMSK, TIFR. А у более крутых AVR, таких как ATMega128, есть еще ETIFR и ETIMSK — своего рода продолжение, так как таймеров там поболее будет.

TIMSK это регистр масок. То есть биты, находящиеся в нем, локально разрешают прерывания. Если бит установлен, значит конкретное прерывание разрешено. Если бит в нуле, значит данное прерывание накрывается тазиком. По дефолту все биты в нуле.

На данный момент нас тут интересуют только прерывания по переполнению. За них отвечают биты

- TOIE0 — разрешение на прерывание по переполнению таймера 0
- TOIE1 — разрешение на прерывание по переполнению таймера 1
- TOIE2 — разрешение на прерывание по переполнению таймера 2

О остальных фичах и прерываниях таймера мы поговорим попозжа, когда будем разбирать ШИМ.

Регистр TIFR это непосредственно флаговый регистр. Когда какое то прерывание срабатывает, то выскакивает там флаг, что у нас есть прерывание. Этот флаг сбрасывается аппаратно когда программа уходит по вектору. Если прерывания запрещены, то флаг так и будет стоять до тех пор пока прерывания не разрешат и программа не уйдет на прерывание.

Чтобы этого не произошло флаг можно сбросить вручную. Для этого в регистре TIFR в него нужно записать 1!

А теперь похимичим
Ну перекроим программу на работу с таймером. Введем программный таймер. Шарманка так и останется, пускай тикает. А мы добавим вторую переменную, тоже на четыре байта:

```
1 ; RAM =====================================================
2                .DSEG
3 CCNT:   .byte   4
4 TCNT:   .byte   4
```

Теперь у нас счетчик будет тикать не каждую итерацию главного цикла, а строго по переполнению таймера. При этом он станет аж пятибайтным. Т.к. младшим теперь будет счетный регистр таймера. В каждой итерации прерывания от таймера он будет увеличиваться на 1 от загруженного значения.
В это время, главная программа будет сравнивать этот счетчик с предельным значением.

Делаем RJMP на обработчик с вектора.

```
1          .ORG $010
2          RETI                  ; (TIMER1 OVF) Timer/Counter1 Overflow
3          .ORG $012
4          RJMP  Timer0_OV       ; (TIMER0 OVF) Timer/Counter0 Overflow
5          .ORG $014
6          RETI                  ; (SPI,STC) Serial Transfer Complete
```

Добавим обработчик прерывания по переполнению таймера 0, в секцию Interrupt. Так как наш тикающий макрос активно работает с регистрами и портит флаги, то надо это дело все сохранить в стеке сначала:

Кстати, давайте создадим еще один макрос, пихающий в стек флаговый регистр SREG и второй — достающий его оттуда.

```
1                .MACRO PUSHF
2                PUSH    R16
3                IN      R16,SREG
4                PUSH    R16
5                .ENDM
6
7
8                .MACRO POPF
9                POP     R16
10               OUT     SREG,R16
11               POP     R16
12               .ENDM
```

Как побочный эффект он еще сохраняет и R16, помним об этом :)

```
1  Timer0_OV:    PUSHF
2                PUSH    R17
3                PUSH    R18
4                PUSH    R19
5
6                INCM    TCNT
7
```

```
8                POP     R19
9                POP     R18
10               POP     R17
11               POPF
12
13               RETI
```

Теперь инициализация таймера. Добавь ее в секцию инита локальной периферии (Internal Hardware Init).

```
1  ; Internal Hardware Init  ====================================
2          SETB    DDRD,4,R16              ; DDRD.4 = 1
3          SETB    DDRD,5,R16              ; DDRD.5 = 1
4          SETB    DDRD,7,R16              ; DDRD.7 = 1
5
6          SETB    PORTD,6,R16             ; Вывод PD6 на вход с подтягом
7          CLRB    DDRD,6,R16              ; Чтобы считать кнопку
8
9          SETB    TIMSK,TOIE0,R16         ; Разрешаем прерывание таймера
10
11         OUTI    TCCR0,1<<CS00           ; Запускаем таймер. Предделитель=1
12                                         ; Т.е. тикаем с тактовой частотой.
13
14         SEI                             ; Разрешаем глобальные прерывания
15 ; End Internal Hardware Init ================================
```

Осталось переписать наш блок сравнения и пересчитать число. Теперь все просто, один тик один такт. Без всяких заморочек с разной длиной кода. Для одной секунды на 8Мгц должно быть сделано 8 миллионов тиков. В хексах это 7A 12 00 с учетом, что младший байт у нас TCNT0, то на наш счетчик остается 7A 12 ну и еще старшие два байта 00 00, их можно не проверять. Маскировать не нужно, таймер мы потом переустановим все равно.

Одна только проблема — младший байт, тот что в таймере. Он тикает каждый такт и проверить его на соответствие будет почти невозможно. Т.к. малейшее несовпадение и условие сравнение выпадет в NoMatch, а подгадать так, чтобы проверка его значения совпала именно с этим тактом… Проще иголку из стога сена вытащить с первой попытки наугад.

Так что точность и в этом случае ограничена — надо успеть проверить значение до того как оно уйдет из диапазона. В данном случае диапазон будет, для простоты, 255 — величина младшего байта, того, что в таймере.

Тогда наша секунда обеспечивается с точностью 8000 000 плюс минус 256 тактов. Не велика погрешность, всего 0,003%.

```
1  ; Main =======================================================
2  Main:           SBIS    PIND,6          ; Если кнопка нажата - переход
3                  RJMP    BT_Push
4
5                  SETB    PORTD,5 ; Зажгем LED2
6                  CLRB    PORTD,4 ; Погасим LED1
7
8  Next:           LDS     R16,TCNT        ; Грузим числа в регистры
9                  LDS     R17,TCNT+1
10
11                 CPI     R16,0x12        ; Сравниванем побайтно. Первый байт
12                 BRCS    NoMatch ; Если меньше -- значит не натикало.
13                 CPI     R17,0x7A        ; Второй байт
14                 BRCS    NoMatch ; Если меньше -- значит не натикало.
15
16 ; Если совпало то делаем экшн
17 Match:          INVB    PORTD,7,R16,R17 ; Инвертировали LED3
18
19 ; Теперь надо обнулить счетчик, иначе за эту же итерацию главного цикла
20 ; мы сюда попадем еще не один раз -- таймер то не успеет натикать 255 значений,
```

```
21 ; чтобы число в первых двух байтах счетчика изменилось и условие сработает.
22 ; Конечно, можно обойти это доп флажком, но проще сбросить счетчик :)
23
24                CLR     R16                     ; Нам нужен ноль
25
26                CLI                             ; Доступ к многобайтной переменной
27                                                ; одновременно из прерывания и фона
28                                                ; нужен атомарный доступ.  Запрет
29 прерываний
30
31                OUTU    TCNT0,R16               ; Ноль в счетный регистр таймера
32                STS     TCNT,R16                ; Ноль в первый байт счетчика в RAM
33                STS     TCNT+1,R16              ; Ноль в второй байт счетчика в RAM
34                STS     TCNT+2,R16              ; Ноль в третий байт счетчика в RAM
35                STS     TCNT+3,R16              ; Ноль в первый байт счетчика в RAM
36                SEI                             ; Разрешаем прерывания снова.
37
38 ; Не совпало - не делаем :)
39 NoMatch:       NOP
40
41                INCM    CCNT            ; Счетчик циклов по тикает
42                                        ; Пускай, хоть и не используется.
43                JMP     Main
44
45
46 BT_Push:       SETB    PORTD,4 ; Зажгем LED1
47                CLRB    PORTD,5 ; Погасим LED2
48
49                RJMP    Next
   ; End Main ================================================
```

Скачать проект с этим примером [1]

Вот как это выглядит в работе

А если надо будет помигать вторым диодиком с другим периодом, то мы смело можем влепить в программу еще одну переменную, а в обработчике прерывания таймера инкрементировать сразу две пееременных. Проверяя их по очереди в главном цикле программы.

Можно еще немного оптимизировать процесс проверки. Сделать его более быстрым.

Надо только сделать счет не на повышение, а на понижение. Т.е. загружаем в переменную число и начинаем его декрементировать в прерывании. И там же, в обработчике, проверяем его на ноль. Если ноль, то выставляем в памяти флажок. А наша фоновая программа этот флажок ловит и запускает экшн, попутно переустанавливая выдержку.

А что если надо точней? Ну тут вариант только один — заюзать обработку события прям в обработчике прерывания, а значение в TCNT:TCNT0 каждый раз подстраивать так, чтобы прерывание происходило точно в нужное время.

AVR. Учебный курс. Использование ШИМ

Вот уже несколько раз я ругался странным словом **ШИМ**. Пора бы внести ясность и разьяснить что же это такое. Вообще, я уже расписывал этот режим работы [1], но все же повторюсь в рамках своего курса.

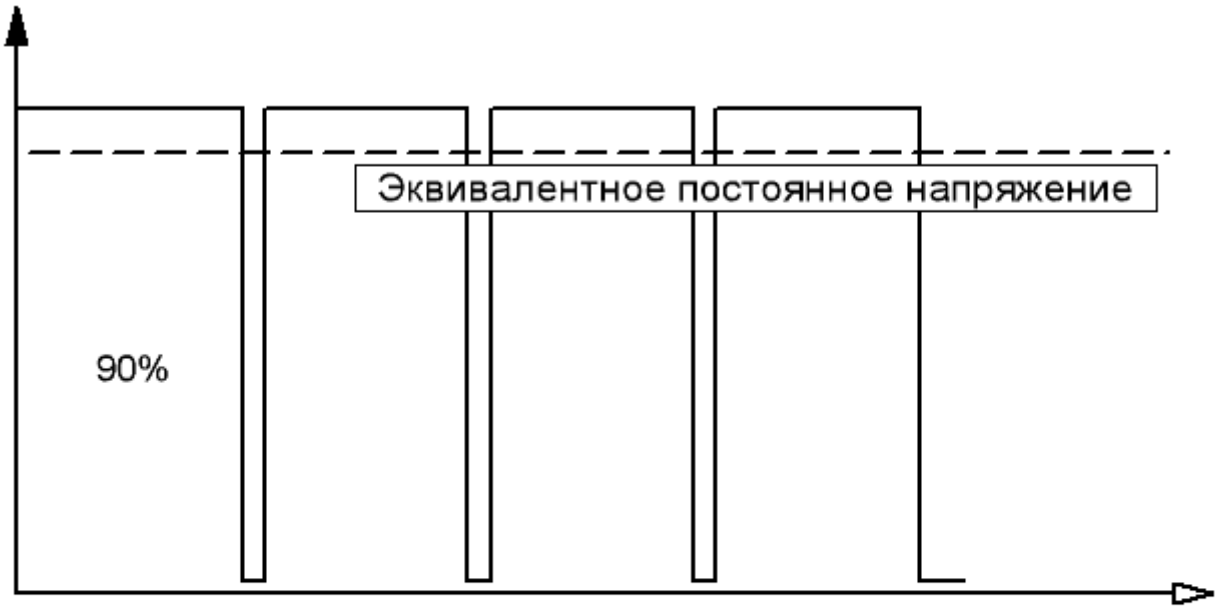
Вкратце, **Широтно Импульсная Модуляция** (в буржуйской нотации этот режим зовется **PWM** — **Pulse Width Modulation**) это способ задания аналогового сигнала **цифровым методом**, то есть из цифрового выхода, дающего только нули и единицы получить какие то плавно меняющиеся величины. Звучит как бред, но тем не менее работает. А суть в чем:

Представь себе тяжеленный маховик который ты можешь вращать двигателем. Причем двигатель ты можешь либо включить, либо выключить. Если включить его постоянно, то маховик раскрутится до максимального значения и так и будет крутиться. Если выключить, то остановится за счет сил трения.

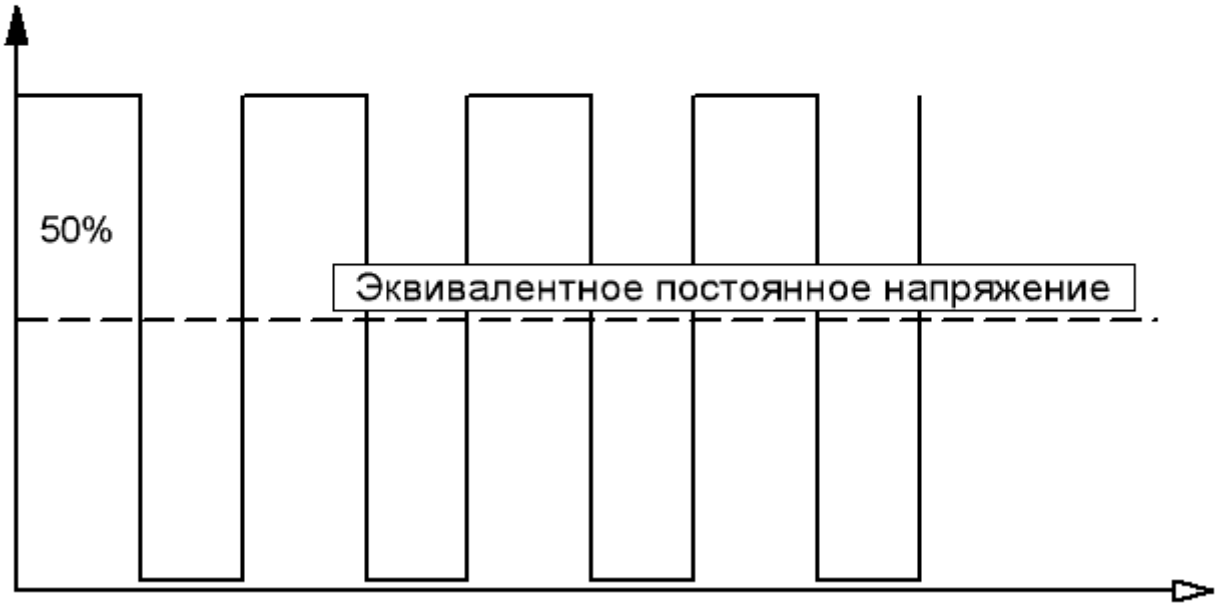
А вот если двигатель включать на десять секунд каждую минуту, то маховик раскрутится, но далеко не на полную скорость — большая инерция сгладит рывки от включающегося двигателя, а сопротивление от трения не даст ему крутится бесконечно долго.

Чем больше **продолжительность включения** двигателя в минуту, тем быстрей будет крутится маховик.
При **ШИМ** мы гоним на выход сигнал состоящий из высоких и низких уровней (применимо к нашей аналогии — включаем и выключаем двигатель), то есть нулей и единицы. А затем это все пропускается через интегрирующую цепочку (в аналогии — маховик). В результате интегрирования на выходе будет величина напряжения, равная площади под импульсами.

Меня **скважность** (отношение длительности периода к длительности импульса) можно плавно менять эту площадь, а значит и напряжение на выходе. Таким образом если на выходе сплошные 1, то на выходе будет напряжение высокого уровня, в случае моего робота, на выходе из моста **L293** это 12 вольт, если нули, то ноль. А если 50% времени будет высокий уровень, а 50% низкий то 6 вольт. Интегрирующей цепочкой тут будет служить масса якоря двигателя, обладающего довольно большой инерцией.

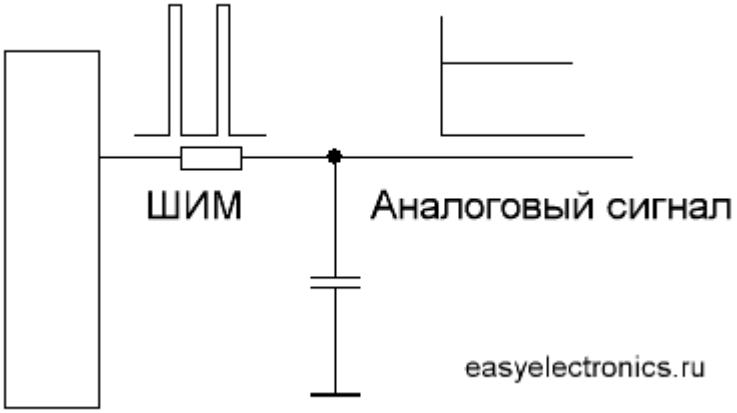

Эквивалентное постоянное напряжение
90%


10%
Эквивалентное постоянное напряжение


50%
Эквивалентное постоянное напряжение

А что будет если взять и гнать **ШИМ** сигнал не от нуля до максимума, а от минуса до плюса. Скажем от +12 до -12. А можно задавать переменный сигнал! Когда на входе ноль, то на выходе -12В, когда один, то +12В. Если скважность 50% то на выходе 0В. Если скважность менять по синусоидальному закону от максимума к минимуму, то получим… правильно! Переменное напряжение. А если взять три таких ШИМ генератора и гнать через них синусоиды сдвинутые на 120 градусов между собой, то получим самое обычное трехфазное напряжение, а значит привет **бесколлекторные асинхронные и синхронные двигатели** — фетиш всех авиамоделистов. На этом принципе построены все современные промышленные привода переменного тока. Всякие **Unidrive** и **Omron Jxx**

В качестве сглаживающей интегрирующей цепи в ШИМ может быть применена обычная RC цепочка:



Так, принцип понятен, приступаем к реализации.
ШИМ сигнал можно сварганить и на операционных усилителях и на микроконтроллере. Причем последние умеют это делать просто мастерски, благо все у них для этого уже есть.

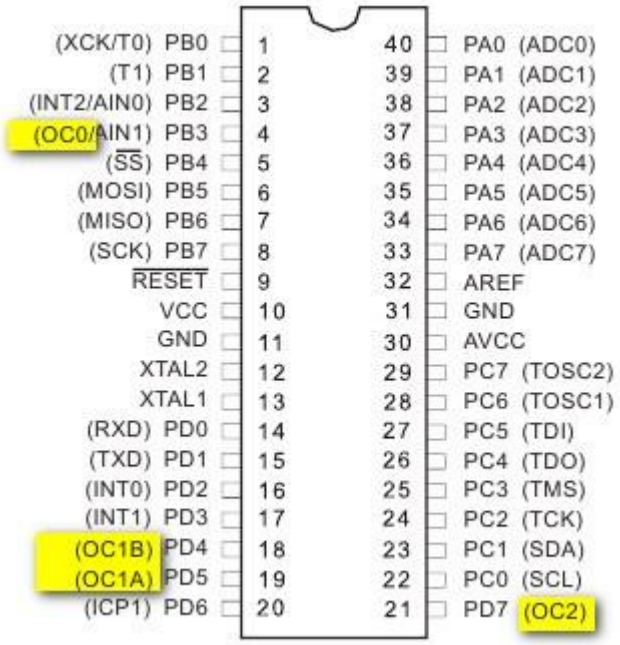
Аппаратный ШИМ

В случае **ATMega16** проще всего сделать на его ШИМ генераторе, который встроен в таймеры. Причем в первом таймере у нас целых два канала. Так что без особого напряга ATmega16 может реализовать одновременно четыре канала **ШИМ**.

Как это реализовано
У таймера есть особый регистр сравнения **OCR****. Когда значение в счётном регистре таймера достигнает значения находящегося в регистре сравнения, то могут возникнуть следующие аппаратные события:

- Прерывание по совпадению
- Изменение состояния внешнего выхода сравнения **OC****.

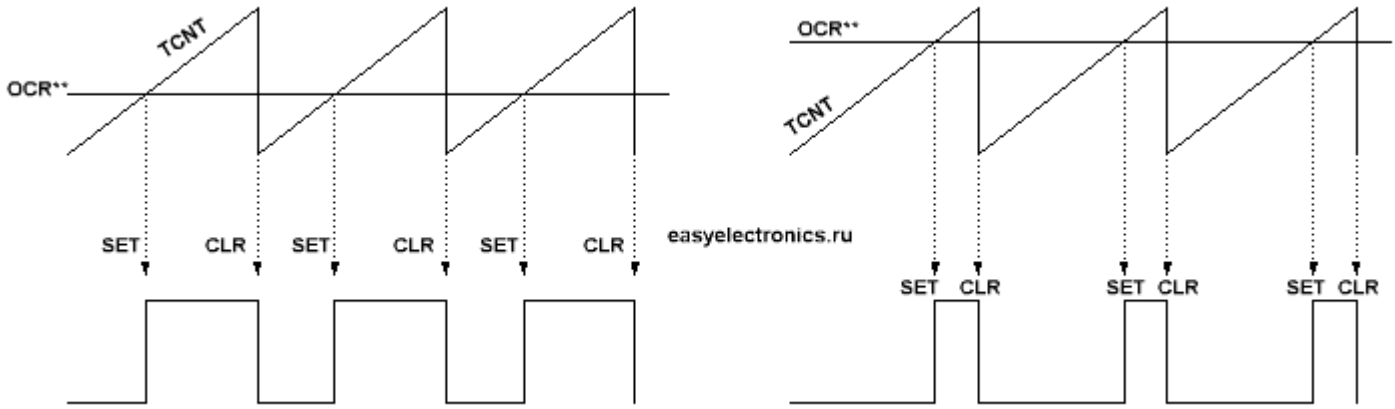
Выходы сравнения выведены наружу, на выводы микроконтроллера



На демоплате Pinboard [2] к этим выводам как раз подключены светодиоды. А если поставить джамперы вдоль, в сторону надписи RC то к выводу ШИМ будет подключена интегрирующая цепочка.

Предположим, что мы настроили наш ШИМ генератор так, чтобы когда значение в счетном регистре больше чем в регистре сравнения, то на выходе у нас 1, а когда меньше, то 0.

Что при этом произойдет? Таймер будет считать как ему и положено, от нуля до 256, с частотой которую мы настроим битами предделителя таймера. После переполнения сбрасывается в 0 и продолжает заново.



Как видишь, на выходе появляются импульсы. А если мы попробуем увеличить значение в регистре сравнения, то ширина импульсов станет уже.

Так что меняя значение в регистре сравнения можно менять скважность ШИМ сигнала. А если пропустить этот ШИМ сигнал через сглаживающую RC цепочку (интегратор) то получим аналоговый сигнал.

У таймера может быть сколько угодно регистров сравнения. Зависит от модели МК и типа таймера. Например, у Атмега16

- Timer0 — один регистр сравнения
- Timer1 — два регистра сравнения (16ти разрядных!)
- Timer2 — один регистр сравнения

Итого — четыре канала. В новых AVR бывает и по три регистра сравнения на таймер, что позволяет одним МК организовать просто прорву независимых ШИМ каналов.

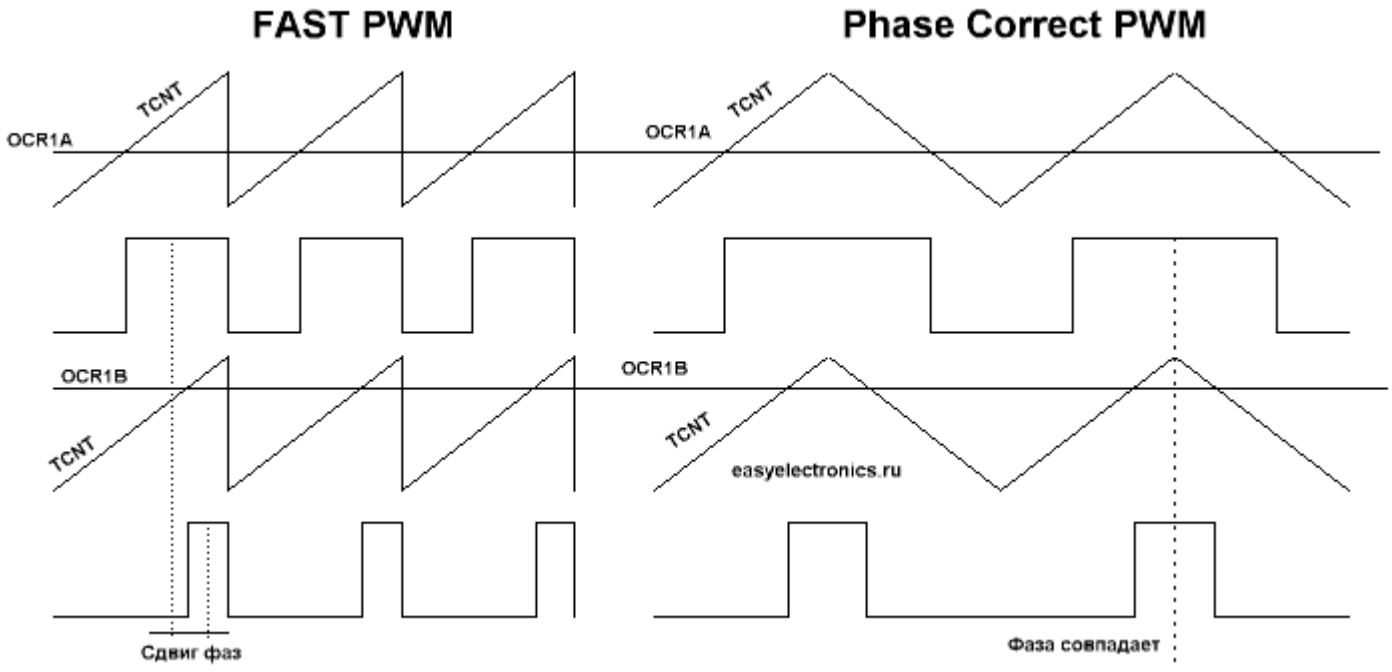
Самих режимов ШИМ существует несколько:

Fast PWM
В этом режиме счетчик считает от нуля до **255**, после достижения переполнения сбрасывается в нуль и счет начинается снова. Когда значение в счетчике достигает значения регистра сравнения, то соответствующий ему вывод **OCxx** сбрасывается в ноль. При обнулении счетчика этот вывод устанавливается в 1. И все!

Частота получившегося ШИМ сигнала определяется просто: Частота процесора 8Мгц, таймер тикает до 256 с тактовой частотой. Значит один период ШИМ будет равен 8000 000/256 = 31250Гц. Вполне недурно. Быстрей не получится — это максимальная скорость на внутреннем **8Мгц** тактовом генераторе. Но если переключить FUSE биты на внешний кварц то можно раскачать МК на 16Мгц.

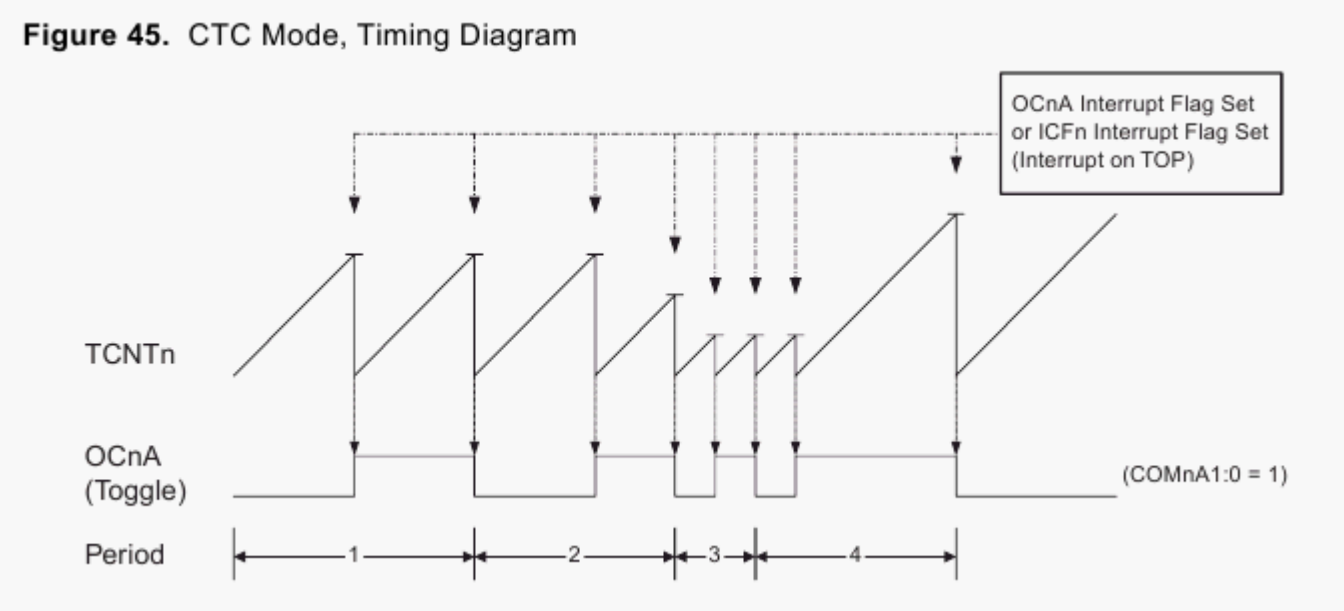
Еще есть возможность повысить разрешение, сделав счет 8, 9, 10 разрядным (если разрядность таймера позволяет), но надо учитывать, что повышение разрядности, вместе с повышением дискретности выходного аналогового сигнала, резко снижает частоту ШИМ.

Phase Correct PWM
ШИМ с точной фазой. Работает похоже, но тут счетчик считает несколько по другому. Сначала от 0 до 255, потом от 255 до 0. Вывод OCxx при первом совпадении сбрасывается, при втором устанавливается.
Но частота **ШИМ** при этом падает вдвое, изза большего периода. Основное его предназначение, делать многофазные ШИМ сигналы, например, трехфазную синусоиду. Чтобы при изменении скважности не сбивался угол фазового сдвига между двумя ШИМ сигналами. Т.е. центры импульсов в разных каналах и на разной скважности будут совпадать.



Еще одна тонкость:
Чтобы не было кривых импульсов, то в регистр сравнения любое значение попадает через буфферный регистр и заносится только тогда, когда значение в счетчике достигнет максимума. Т.е. к началу нового периода **ШИМ** импульса.

Clear Timer On Compare
Сброс при сравнении. Это уже скорей ЧИМ — частотно-импульсно моделированный сигнал. Тут работает несколько иначе, чем при других режимах. Тут счетный таймер тикает не от 0 до предела, а от 0 до регистра сравнения! А после чего сбрасывается.



В результате, на выходе получаются импульсы всегда одинаковой скважности, но разной частоты. А чаще всего этот режим применяется когда надо таймером отсчитывать периоды (и генерить прерывание) с заданной точностью.

Например, надо нам прерывание каждую миллисекунду. И чтобы вот точно. Как это реализовать проще? Через Режим CTC! Пусть у нас частота 8Мгц.

Прескалер будет равен 64, таким образом, частота тиков таймера составит 125000 Гц. А нам надо прерывание с частотой 1000Гц. Поэтому настраиваем прерывание по совпадению с числом 125.

Дотикал до 125 — дал прерывание, обнулился. Дотикал до 125 — дал прерывание, обнулился. И так бесконечно, пока не выключим.

Вот вам и точная тикалка.

Нет, конечно, можно и вручную. Через переполнение, т.е. дотикал до переполнения, загрузил в обработчике прерывания заново нужные значение TCNTx=255-125, сделал нужные полезные дела и снова тикать до переполнения. Но ведь через CTC красивей! :)

Аппаратура
А теперь контрольные регистры, которыми все это безобразие задается и программируется. Опишу на примере Двухканального FastPWM на таймере 1. В других все похоже. Даташит в зубы и вперед.

Итак, тут правят бал регистры **TCCR1A** и **TCCR1B**. Гы, кто бы сомневался %)

Распишу их по битам.
Регистр **TCCR1A**, биты **COM1A1:COM1A0** и **COM1B1:COM1B0**. Эта братия определяет поведение вывода сравнения **OC1A** и **OC1B** соответственно.

COMxx1	COMxx0	Режим работы выхода

0	0	вывод отцеплен от регистра сравнения и не меняется никак.
0	1	Поведение вывода зависит от режима заданного в WGM, различается для разных режимов (FastPWM, FC PWM, Compar out) и разных МК, надо сверяться с даташитом.
1	0	прямой ШИМ (сброс при совпадении и установка при обнулении счета)
1	1	обратный ШИМ (сброс при обнулении и установка при совпадении)

Регистр **TCCR1A**, биты **WGM11** и **WGM10** вместе с битами **WGM12** и **WGM13**, находящимися в регистре TCCR1B задают режим работы генератора.

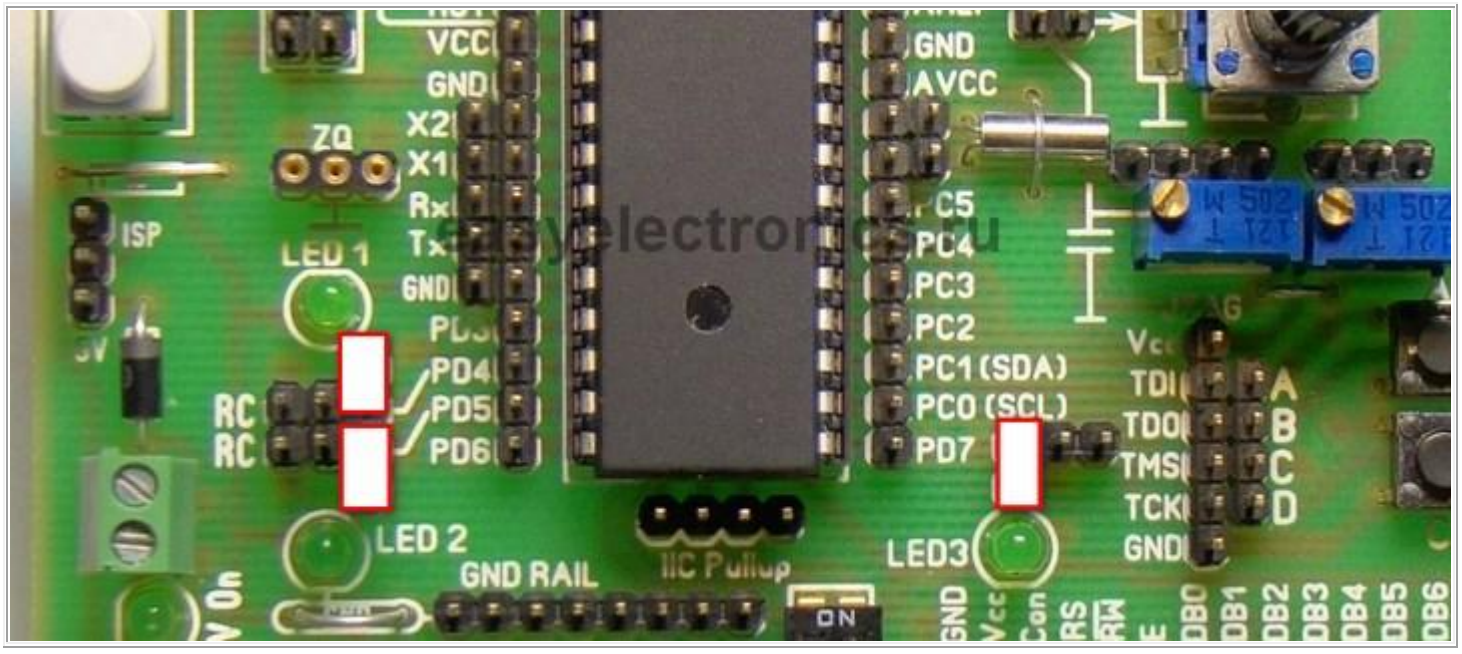
WGM13	WGM12	WGM11	WGM10	Режим работы
0	1	0	1	Fast PWM 8 бит
0	1	1	0	Fast PWM 9 бит
0	1	1	1	Fast PWM 10 бит

Другие комбинации битов **WGM** задают режимы P**hase Correct PWM** и **CTC** (сброс **OCxx** при совпадении). Если интересно, то читай даташит, я для себя много интересного там не нашел, кроме **Phase Correct PWM**. И то мне сейчас важней скорость, а не точность фазы :)

После остается только запустить таймер, установив бит **CS10** (подсчет тактовых импульсов **с делителем 1:1**)

Пример кода:

Попробуем поиграться яркостью светодиодов с помощью ШИМ сигналов. Подключи джамперы, чтобы запитать светодиоды LED1 и LED2



Теперь все готово, можно писать код. Вначале в раздел инициализации устройств добавляю настройку таймера на запуск ШИМ и подготовку выводов.

```
1  ;FastPWM Init
2          SETB    DDRD,4,R16      ; DDRD.4 = 1 Порты на выход
3          SETB    DDRD,5,R16      ; DDRD.5 = 1
4
5  ; Выставляем для обоих каналов ШИМ режим вывода OC** сброс при совпадении.
6  ; COM1A = 10 и COM1B = 10
```

```
7  ; Также ставим режим FAST PWM 8bit (таймер 16ти разрядный и допускает
8  ; большую разрядность ШИМ сигнала. Вплоть до 10 бит.  WGM = 0101
9  ; Осталось только запустить таймер на частоте МК CS = 001
10
11         OUTI    TCCR1A,2<<COM1A0|2<<COM1B0|0<<WGM11|1<<WGM10
12         OUTI    TCCR1B,0<<WGM13|1<<WGM12|1<<CS10
```

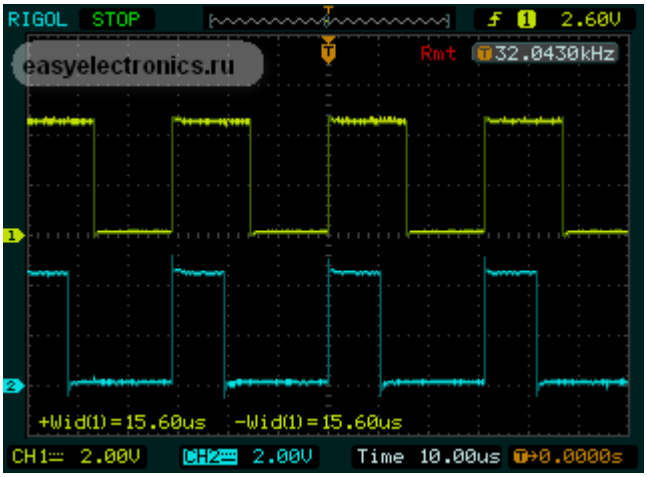
Готово! Теперь ШИМ таймера1 генерит сигнал на выходах OC1A и OC1B

Закинем в регистры сравнения первого и второго канала число 255/3=85 и 255/2 = 128
Так как ШИМ у нас 8ми разрядный, то заброс идет только в младший разряд. Старший же остается нулем. Но регистры сравнения тут у нас 16ти разрядные поэтому грузить надо оба байта сразу. Не забыв запретить прерывания (это важно!!! ибо атомарный доступ)

```
1          CLI
2          OUTI    OCR1AH,0
3          OUTI    OCR1AL,85
4
5          OUTI    OCR1BH,0
6          OUTI    OCR1BL,128
7          SEI
```

Поехали! :)

Прошиваем, тыкаемся в ноги микроконтроллера осциллографом — видим следующую картину по каналам:



Как мы и запланировали. С первого канала длительность импульса в 1/3 периода, а со второго в 1/2
Ну и светодиоды горят с разной яркостью. Один ярче, другой тусклей. Меняя значение в регистрах OCR*** мы можем менять скважность.

Давай сделаем так, чтобы светодиод плавно менял свою яркость от нуля до максимума. Как помнишь, у нас там была программа, с мигающем по таймеру0 светодиодом. Немного ее подправим, сделаем так, чтобы по таймеру не светодиод мигал, а менялось значение в регистрах сравнения OCR1A и OCR1B. Причем меняться оно будет в разные стороны :)

```
1  ; Main ============================================================
2  Main:   LDS     R16,TCNT        ; Грузим числа в регистры
3          LDS     R17,TCNT+1
4
5          CPI     R16,0x10        ; Сравниванем побайтно выдержку
6          BRCS    NoMatch
7          CPI     R17,0x01        ; Выдержку сделали поменьше = 0x0110
8          BRCS    NoMatch
9
```

```
10 ; Если совпало то делаем экшн
11 Match:  CLI                    ; Запрет прерываний, т.к. атомарный доступ
12
13 ; Меняем первый канал
14 ; Особенность 16ти разрядных регистров в том, что их надо правильно читать и записывать.
15 ; Читают вначале младший, потом старший байты. Так надо, чтобы младший не успел
16 измениться
17 ; (он ведь может тикать по таймеру) пока читают первым старший.  Укладывают их в
18 обратном
19 ; порядке. Сначала старший, потом младший. Правда для регистров OCR это не имеет большой
20 ; разницы -- они статичные, а вот для TCNT очень даже!
21
22         IN      R16,OCR1AL      ; Достали первый байт сравнения
23         IN      R17,OCR1AH      ; он 16ти разрядный, но старший байт будет 0
24
25         INC     R16             ; Увеличили
26
27         OUT     OCR1AH,R17      ; И сунули их обратно
28         OUT     OCR1AL,R16
29
30 ; Меняем второй канал
31         IN      R16,OCR1BL      ; Достали второй байт сравнения
32         IN      R17,OCR1BH      ; он 16ти разрядный, но старший байт будет 0
33
34         DEC     R16             ; Уменьшили
35
36         OUT     OCR1BH,R17      ; И сунули их обратно
37         OUT     OCR1BL,R16
38         SEI                     ; Конец атомарного доступа
39
40 ; Теперь надо обнулить счетчик, иначе за эту же итерацию главного цикла
41 ; Мы сюда попадем еще не один раз -- таймер то не успеет натикать 255 значений
42 ; чтобы число в первых двух байтах счетчика изменилось.
43
44         CLR     R16             ; Нам нужен ноль
45         CLI                     ; Таймер меняется и в прерывании. Нужен
46                                 ; атомарный доступ. Запрещаем прерывания
47         OUT     TCNT0,R16       ; Ноль в счетный регистр таймера
48         STS     TCNT,R16        ; Ноль в первый байт счетчика в RAM
49         STS     TCNT+1,R16      ; Ноль в второй байт счетчика в RAM
50         STS     TCNT+2,R16      ; Ноль в третий байт счетчика в RAM
51         STS     TCNT+3,R16      ; Ноль в первый байт счетчика в RAM
52         SEI                     ; Разрешаем прерывания.
53 ; Не совпало - не делаем :)
54 NoMatch:        NOP
55
           INCM    CCNT            ; Шарманка вращается дальше, вхолостую
           JMP     Main
```

А теперь давайте включим режим с точной фазой (WGM = 0001) и посмотрим на то как будет меняться скважность.

```
1         OUTI    TCCR1A,2<<COM1A0|2<<COM1B0|0<<WGM11|1<<WGM10
2         OUTI    TCCR1B,0<<WGM13|0<<WGM12|1<<CS10
```

ШИМ на прерываниях.
Но вот засада — плата уже разведена, захотелось **ШИМ**, а выводы **OCxx** уже задействованы под другие цели.

Ничего страшного, малой кровью можно это исправить. Также запускаем **ШИМ**, только:

- Отключаем выводы OCxx от регистра сравнения.

- Добавляем два обработчика прерывания на сравнение и на переполнение. В прерывании по сравнению сбрасываем нужный бит, в прерывании по переполнению счетчика устанавливаем.

Все просто :)

Пример:

```
1  ;FastPWM Init на прерываниях
2
3  ; ШИМ будет на выводах 3 и 6 порта D
4          SETB    DDRD,3,R16      ; DDRD.3 = 1 Порты на выход
5          SETB    DDRD,6,R16      ; DDRD.6 = 1
6
7  ; Выставляем для обоих каналов ШИМ режим вывода OC** выключеным.
8  ; COM1A = 00 и COM1B = 00
9  ; Также ставим режим FAST PWM 8bit (таймер 16ти разрядный и допускает
10 ; большую разрядность ШИМ сигнала. Вплоть до 10 бит.  WGM = 0101
11 ; Осталось только запустить таймер на частоте МК CS = 001
12
13         OUTI    TCCR1A,0<<COM1A0|0<<COM1B0|0<<WGM11|1<<WGM10
14         OUTI    TCCR1B,0<<WGM13|1<<WGM12|1<<CS10
15
16         SETB    TIMSK,OCIE1A,R16        ; Включаем прерывание по сравнению А
17         SETB    TIMSK,OCIE1B,R16        ; Включаем прерывание по сравнению Б
18         SETB    TIMSK,TOIE1,R16; Включаем прерывание по переполнению Т1
19                                         ; Причем в режиме WGM=1010 переполнение
20                                         ; будет на FF т.е. таймер работает как
21                                         ; 8ми разрядный.
```

Осталось только прописать обработчики и вектора:

```
1           .CSEG
2           .ORG $000               ; (RESET)
3           RJMP   Reset
4           .ORG $002
5           RETI                    ; (INT0) External Interrupt Request 0
6           .ORG $004
7           RETI                    ; (INT1) External Interrupt Request 1
8           .ORG $006
9           RETI                    ; (TIMER2 COMP) Timer/Counter2 Compare Match
10          .ORG $008
11          RETI                    ; (TIMER2 OVF) Timer/Counter2 Overflow
12          .ORG $00A
13          RETI                    ; (TIMER1 CAPT) Timer/Counter1 Capture Event
14          .ORG $00C
15          RJMP Timer1_OCA                 ; (TIMER1 COMPA) Timer/Counter1 Compare Match A
16          .ORG $00E
17          RJMP Timer1_OCB                 ; (TIMER1 COMPB) Timer/Counter1 Compare Match B
18          .ORG $010
19          RJMP Timer1_OVF                 ; (TIMER1 OVF) Timer/Counter1 Overflow
20          .ORG $012
21          RJMP  Timer0_OV         ; (TIMER0 OVF) Timer/Counter0 Overflow
22          .ORG $014
23          RETI                    ; (SPI,STC) Serial Transfer Complete
24          .ORG $016
25          RETI                            ; (USART,RXC) USART, Rx Complete
26          .ORG $018
27          RETI                    ; (USART,UDRE) USART Data Register Empty
28          .ORG $01A
29          RETI                    ; (USART,TXC) USART, Tx Complete
30          .ORG $01C
31          RETI                    ; (ADC) ADC Conversion Complete
```

```
32          .ORG $01E
33          RETI                    ; (EE_RDY) EEPROM Ready
34          .ORG $020
35          RETI                    ; (ANA_COMP) Analog Comparator
36          .ORG $022
37          RETI                    ; (TWI) 2-wire Serial Interface
38          .ORG $024
39          RETI                    ; (INT2) External Interrupt Request 2
40          .ORG $026
41          RETI                    ; (TIMER0 COMP) Timer/Counter0 Compare Match
42          .ORG $028
43          RETI                    ; (SPM_RDY) Store Program Memory Ready
44
45          .ORG   INT_VECTORS_SIZE       ; Конец таблицы прерываний
46
47 ; Interrupts ============================================
48 Timer0_OV:      PUSHF
49                 PUSH    R17
50                 PUSH    R18
51                 PUSH    R19
52
53                 INCM    TCNT
54
55                 POP     R19
56                 POP     R18
57                 POP     R17
58                 POPF
59
60                 RETI
61
62 ; Вот наши обработчики на ШИМ
63 Timer1_OCA:     SBI     PORTD,3
64                 RETI
65
66 Timer1_OCB:     SBI     PORTD,6
67                 RETI
68
69 Timer1_OVF:     CBI     PORTD,3
70                 CBI     PORTD,6
71                 RETI
72 ; End Interrupts ========================================
```

Почему я в этих обработчиках не сохраняю регистры и SREG? А незачем! Команды SBI меняют только конкретные биты (а больше нам и не надо), не влияя на флаги и другие регистры.

Запустили…

И получили полную херню. Т.е. ШИМ как бы есть, но почему то адово мерцает. А на осциллографе в этот момент полный треш. Кто виноват? Видимо конфликт прерываний. Осталось только выяснить где именно. Сейчас я вам дам практический пример реалтаймовой отладки :)

Итак, что мы имеем:

ШИМ, как таковой, работает. Скважность меняется. Значит наш алгоритм верен.
Но длительности скачут. Почему? Видимо потому, что что-то мешает им встать вовремя. Когда у нас возникают фронты? Правильно — по прерываниям. А прерывания по таймерам. Т.е. врать не должны. Однако так получается. Давайте узнаем каком месте у нас конфликт.

Первым делом надо добавить в код обработчика отладочную инфу. Будем в обработчике прерываний инвертировать бит. Пусть это будет PD7 — зашли в обработчик, инверснули. Зашли — инверснули. В результате, у нас на выходе этого бита будет прямоугольный сигнал, где каждый фронт — сработка прерываний. Послужит нам как линейка, отмеряющая время.

```
1  ; Interrupts ============================================
2  Timer0_OV:      PUSHF
3                  PUSH    R17
4                  PUSH    R18
5                  PUSH    R19
6
7                  INCM    TCNT
8
9                  POP     R19
10                 POP     R18
11                 POP     R17
12                 POPF
13
14                 RETI
15
16 ; Установка бита ШИМ канала А
17 Timer1_OCA:     SBI     PORTD,3
18                 RETI
19
20 ; Установка бита ШИМ канала Б
21 Timer1_OCB:     SBI     PORTD,6
22                 RETI
23
24 ;Сброс бита ШИМ канала А и Б
25 Timer1_OVF:     CBI     PORTD,3
26                 CBI     PORTD,6
27
28 ;DEBUG PIN BEGIN ---------------
29                 PUSHF
30                 INVBM   PORTD,7
31                 POPF
32 ;DEBUG PIN END -----------------
33                 RETI
```

Инверсия бита невозможна без логических операций, поэтому надо сохранять флаги.

Из картинки стало понятно, что у нас накрывается прерывание по сравнению. Давайте попробуем посмотреть с какими прерыванием происходит конфликт. Особых вариантов у нас нет — прерываний у нас тут четрые. А наиболее очевиден конфликт Timer0_OV vs Timer1_OCA vs Timer1_OCB.

OCA и OCB конфликтуют только тогда, когда счетные регистры у них сравниваются — вызов происходит почти одновременно, но сами обработчики короткие — всего несколько тактов, поэтому дребезг не столь сильный.

А вот Timer0_OV делает довольно мощный прогруз стека и еще вычитает четырехбайтную переменную. Т.е. тактов на 20 может задержать обработчик установки бита Timer1_OC* от того и вылазят такие зверские дребезги.

Давайте проверим эту идею. Разрешим прерывания в обработчике Timer0_0V

```
1  ; Interrupts ============================================
2  Timer0_OV:      SEI
3                  PUSHF
4                  PUSH    R17
5                  PUSH    R18
6                  PUSH    R19
7
8                  INCM    TCNT
9
10                 POP     R19
11                 POP     R18
12                 POP     R17
13                 POPF
14
```

```
15                RETI
16
17 ; Установка бита ШИМ канала А
18 Timer1_OCA:     SBI     PORTD,3
19                RETI
20
21 ; Установка бита ШИМ канала Б
22 Timer1_OCB:     SBI     PORTD,6
23                RETI
24
25 ;Сброс бита ШИМ канала А и Б
26 Timer1_OVF:     CBI     PORTD,3
27                CBI     PORTD,6
28                RETI
```

Картина сразу исправилась. Теперь более важное (для нас важное) прерывание задвигает обработчик от Таймера 0. Но тут надо просекать возможные риски:

- Более глубокий прогруз стека
- Нарушается атомарный доступ к четырехбайтной переменной TCNT, поэтому если бы у нас было еще какое-то прерывание, меняющее TCNT то его надо было бы запрещать локально. Иначе бы мы получили такой трешняк, что проще заново прогу переписать, чем это отладить

.

ШИМ на таймерах
Когда совсем все плохо, то можно сделать на любом таймере. В обработчик прерывания по переполнению таймера заносим конечный автомат, который сначала загрузит в таймер длительность низкого уровня, а при следующем заходе — длительность высокого. Ну и, само собой, ноги процессора подергает как надо. Таким образом, на один таймер можно повесить дофига **ШИМ** каналов, но задолбаешься все с кодовой реализацией всего этого. И процессорное время жрать будет некисло. Не говоря уже про дребезги, о которых только что было сказано. Это для эстетов извращенцев :)))))

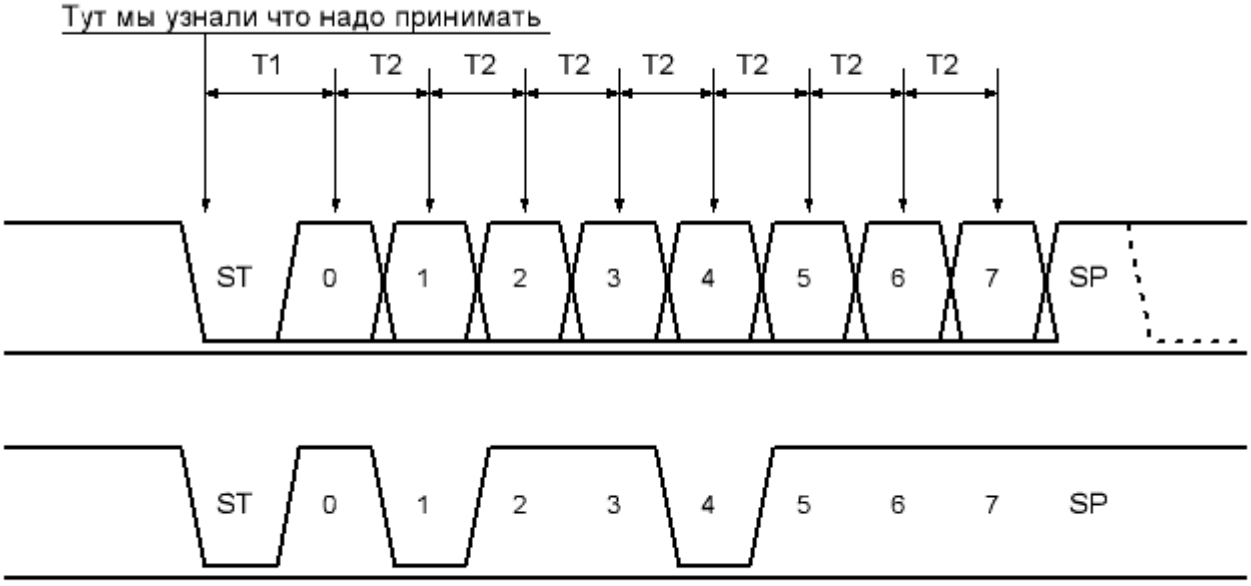
Исходник к статье [3]

AVR. Учебный курс. Передача данных через UART

Почти каждый микроконтроллер имеет на борту универсальный последовательный интерфейс — **UART**. **AVR** тут не исключение и поддерживает этот протокол в полном обьеме полностью аппаратно. По структуре это обычный **асинхронный** последовательный протокол, то есть передающая сторона по очереди выдает в линию 0 и 1, а принимающая отслеживает их и запоминает. Синхронизация идет по времени — приемник и передатчик заранее договариваются о том на какой частоте будет идти обмен. Это очень важный момент! Если скорость передатчика и приемника не будут совпадать, то передачи может не быть вообще, либо будут считаны не те данные.

Протокол
Вначале передатчик бросает линию в низкий уровень — это *старт бит*. Почуяв что линия просела, приемник выжидает интервал T1 и считывает первый бит, потом через интервалы T2 выковыриваются остальные биты. Последний бит это *стоп бит*. Говорящий о том, что передача этого байта завершена. Это в самом простом случае.

В конце байта, перед стоп битом, может быть и бит четности. Который получается если поксорить между собой все биты, для контроля качества передачи. Также может быть два стопа, опять же для надежности. Битов может быть не 8, а 9. О всех этих параметрах договариваются на берегу, до начала передачи. Самым же популярным является 8 бит, один старт один стоп, без четности.

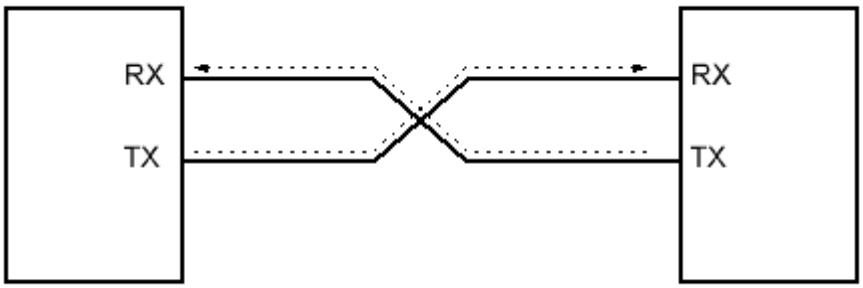


Передача байта 11101101

Причем с самим протоколом можно не заморачиваться — все реализовано аппаратно. Разве что захочется завести второй UART, тогда придется делать его программно.

По такому же протоколу работает COM порт компьютера, разница лишь в разнице напряжений, поэтому именно этот протокол я буду использовать для связи микроконтроллера с компом. Для преобразования напряжений можно использовать RS232-TTL конвертер. [1] Мы же применим встроенный в Pinboard [2] мост USB-UART который образовывает в системе виртуальный COM PORT

Аппаратная часть
Ну тут все просто, соединяем крест-накрест приемник с передатчиком и готово.



Внутри же вначале находится накопительный сдвиговый регистр, в котором происходит сборка байта из битов и регистры данных **UDR**, куда этот бит передается. Такая структура исключает возможность считать не до конца полученный байт.
Буфер приема состоит из двух байт, что позволяет ему держать два в уме и один еще засасывать в сдвиговый регистр.

Использование UART'а

У **AVR** есть регистр **UDR** это UART Data Register (в некоторых контроллерах он может зваться UDR0 или еще как нибудь похоже). На самом деле это два разных регистра, но имеют один адрес. Просто на запись попадает в один

(регистр передатчика), а на чтение берет из другого (регистр приемника). Таким образом достаточно просто пихать данные в этот регистр и они улетят приемнику, и, наоборот, считывать их оттуда по приходу.

О том, что байт пришел в регистр **UDR** нам скажет прерывание по завершению приема, которое вызывается сразу же, как приемник засосет в себя все биты (обычно 8, но может быть и 9, в зависимости от настройки).

Поскольку передача идет довольно медленно, то бездумно пихать данные в регистр **UDR** нельзя — нужно дождаться окончания передачи предыдущего байта. О том, что **UDR** пуст и готов к приему нового байта сигнализирует бит **UDRE**, он же вызовет аппаратное прерывание по опустошению буфера.

Так что постоянно следить за **UART** вручную не нужно, все обслуживание можно повесить на прерывания и он сам будет все делать. Можно в памяти организовать буфер и туда тупо пихать данные, а на прерывании по опустошению **UDR** следить за тем есть ли что в буфере и если есть — отправлять.

А по приему, не тупить, а также, по прерыванию, пихать данные в ОЗУ, но уже в буфер приема, откуда их считает уже программа.

Настройка UART
Все настройки приемопередатчика хранятся в регистра конфигурации. Это **UCSRA, UCSRB и UCSRC**. А скорость задается в паре **UBBRH:UBBRL**.

Досконально все расписывать не буду — на это есть даташит. Напишу лишь то, что жизненно необходимо.

Регистр **UCSRA**

Тут нам интересны только биты **RXC** и **TXC** это флаги завершения приема и передачи, соответственно. RXC встанет когда непрочитанный байт вылезет в регистр UDR, а TXC встает когда последний стоп-бит прошел, а новое значение в UDR не поступило. Т.е. после прохода всех байтов.

Также одновременно с этими флагами вызывается прерывание (если оно было разрешено). Сбрасываются они аппаратно — принимающий после чтения из регистра **UDR**, передающий при переходе на прерывание, либо программно (чтобы сбросить флаг программно в него надо записать 1)

Биты **UDRE** сигнализирует о том, что регистр **UDR** приемника пуст и в него можно пихать новый байт. Сбрасывается он аппаратно после засылки в **UDR** какого либо байта. Также генерируется прерывание «Регистр пуст»

Бит **U2X** — это бит удвоения скорости при работе в ассинхронном режиме. Его надо учитывать при расчете значения в **UBBRH:UBBRL**

Регистр UCSRB
Тут в первую очередь это биты **RXEN** и **TXEN** — разрешение приема и передачи. Стоит их сбросить как выводы UART тут же становятся обычными ножками I/O.

Биты **RXCIE, TXCIE, UDRIE** разрешают прерывания по завершению приема, передачи и опустошении буфера передачи **UDR.**

Регистр UCSRC
Самый прикол тут — это бит селектора **URSEL** дело в том, что по неизвестной причине создаетели решили сэкономить байт адреса и разместили регистры **UCSRC** и **UBRRH** в одной адресном пространстве. А как же определять куда записать? А по старшему биту! Поясню на примере. Если мы записываем число у которого седьмой бит равен 1, то оно попадет в **UCSRC**, а если **0** то **UBRRH**. Причем этот бит есть не во всех AVR в новых моделях его нет, а регистры имеют разные адреса. Так что надо смотреть в даташите этот момент — есть там бит URSEL или нет.

Остальные биты задают число стопов, наличие и тип контроля четности. Если оставить все по дефолту то будет стандартный режим. Надо только выставить формат посылки. Делается это битами **UCSZ0, UCSZ1 и UCSZ2** (этот бит тусуется в регистре **UCSRB**). Для стандартной 8ми битной посылки туда надо записать две единички.

Скорость обмена.
Тут все зависит от пары **UBBRx**

Вычисляется требуемое значение по формуле:

UBBR=XTAL/(16*baudrate)-1 для U2X=0
UBBR=XTAL/(8*baudrate)-1 для U2X=1

Где:
XTAL — рабочая тактовая частота контроллера.
baudrate — требуемая скорость (я люблю 9600 :) — чем медленней тем надежней. 9600 в большинстве случаев хватает)

Ошибки передачи
К сожалению мир наш не идеален, поэтому возможны ошибки при приеме. За них отвечают флаги в регистре **UCSRA**
FE — ошибка кадрирования. Т.е. мы ждали стоп бит, а пришел 0.
OR — переполнение буфера. То есть данные лезут и лезут, а из UDR мы их забирать не успеваем.
PE — не совпал контроль четности.

Примеры кода

Простая приемка и отправка байт. Без использования прерываний.
Для начала инициализация UART в ATMega16 (А таже в Mega8535/8/32 и многих других).

```
1
2  ; Internal Hardware Init  ====================================
3                  .equ    XTAL = 8000000
4                  .equ    baudrate = 9600
5                  .equ    bauddivider = XTAL/(16*baudrate)-1
6
7
8  uart_init:      LDI     R16, low(bauddivider)
9                  OUT     UBRRL,R16
10                 LDI     R16, high(bauddivider)
11                 OUT     UBRRH,R16
12
13                 LDI     R16,0
14                 OUT     UCSRA, R16
15
16 ; Прерывания запрещены, прием-передача разрешен.
17                 LDI     R16, (1<<RXEN)|(1<<TXEN)|(0<<RXCIE)|(0<<TXCIE)|(0<<UDRIE)
18                 OUT     UCSRB, R16
19
20 ; Формат кадра - 8 бит, пишем в регистр UCSRC, за это отвечает бит селектор
21                 LDI     R16, (1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1)
22                 OUT     UCSRC, R16
```

```
  ; Процедура отправки байта
1 uart_snt:       SBIS    UCSRA,UDRE      ; Пропуск если
2 нет флага готовности
3                  RJMP    uart_snt        ; ждем
4 готовности - флага UDRE
5
6                  OUT     UDR, R16        ; шлем байт
                   RET                     ; Возврат
```

```
1 ;Посылка байта:
2                 RCALL   uart_init       ; вызываем нашу процедуру инициализации.
3
4 Main:           LDI     R16,'E'         ; загоняем в регистр код буквы «E»
5                 RCALL   uart_snt        ; Вызываем процедуру отправки байта.
6
7                 NOP                     ; Любой дальнейший код
8                 NOP
9                 NOP
```

```
1 ;Ожидание байта
```

```
2 uart_rcv:       SBIS    UCSRA,RXC       ; Ждем флага прихода байта
3                 RJMP    uart_rcv        ; вращаясь в цикле
4
5                 IN      R16,UDR         ; байт пришел - забираем.
6                 RET                     ; Выходим. Результат в R16
```

В главном цикле это выглядит так (я выбросил из главного цикла все что там было для краткости, чтобы в глазах не рябило. Оставил только работу с USART):

```
1 ; Main ============================================================
2 Main:           RCALL   uart_rcv        ; Ждем байта
3
4                 INC     R16             ; Делаем с ним что-то
5
6                 RCALL   uart_snt        ; Отправляем обратно.
7
8                 JMP     Main
```

Если зальешь эту программку в Pinboard [2] и подключишься терминалкой, то на каждый байт контроллер тебе вернет байт следующий по величине. Например на 1 (код 0×31) возвращается 2 (код 0×32). И так далее.

Данный метод прост и работает в лоб. Но имеет одну большую проблему — цикл ожидания прихода байта. В этот момент ничего работать не может. Конечно, можно не зацикливаться наглухо, а просто проверять флаг RxC и крутиться в основном цикле. Но тогда, если передача идет на большой скорости, можно запросто прозевать байт.

Работа на прерываниях
Решение тут одно — использование прерываний хотя бы на прием. А в идеале и на передачу тоже. Сейчас покажу тебе пример буферизированной работы с приемопередатчиком на прерываниях.

Во первых инициализация, она теперь другая — мы разрешаем прерывания:

```
1  ; Internal Hardware Init  ====================================
2                  .equ    XTAL = 8000000
3                  .equ    baudrate = 9600
4                  .equ    bauddivider = XTAL/(16*baudrate)-1
5
6
7  uart_init:      LDI     R16, low(bauddivider)
8                  OUT     UBRRL,R16
9                  LDI     R16, high(bauddivider)
10                 OUT     UBRRH,R16
11
12                 LDI     R16,0
13                 OUT     UCSRA, R16
14
15 ; Прерывания разрешены, прием-передача разрешен.
16                 LDI     R16, (1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(0<<UDRIE)
17                 OUT     UCSRB, R16
18
19 ; Формат кадра - 8 бит, пишем в регистр UCSRC, за это отвечает бит селектор
20                 LDI     R16, (1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1)
21                 OUT     UCSRC, R16
```

Используем все три прерывания. По приему, по опустошению буфера и по окончании передачи. Но пока разрешаем только два — по приему и по передаче. Иначе при старте мы сразу же ускачем на обработчик UDRIE — буфер то пуст!

В таблицу векторов впишем наши переходы:

```
1          .ORG $016
```

```
2         RJMP    RX_OK           ; (USART,RXC) USART, Rx Complete
3         .ORG $018
4         RJMP    UD_OK           ; (USART,UDRE) USART Data Register Empty
5         .ORG $01A
6         RJMP    TX_OK           ; (USART,TXC) USART, Tx Complete
```

А в секцию обработчиков прерываний добавим нашу процедуру приема:

```
   RX_OK:          PUSHF                   ; Макрос, пихающий в стек SREG и R16
1
2                  IN      R16,UDR         ; Тут главное забрать байт из UDR иначе
3                                          ; флаг прерывания не снимется
4                                          ; Дальше, если время позволяет, можно и обработать
5
6                  CPI     R16,Value       ; Например, разобрать по байтам и выполнить
7  действие
8                  BRNE    NXT             ; Обычным CASE оператором.
9                  Action1                 ; Делаем что нибудь ценное. Главное не забыть в
10 стеке
11                                         ; Регистры попрятать. А то будет тебе упячка.
12 NXT:            CPI     R16,Value2
13                 BRNE    RX_Exit
14                 Action2                 ; Делаем что нибудь ценное2
15
16 Rx_Exit:                POPF                    ; Достаем SREG и R16
                   RETI
```

Передача делается побайтно, на вызовах прерываний. Например, надо нам отрыгнуть текстовую строку из флеша:

```
1 String:         .db     "Hello Interrupt Request",0
```

Заводим указатель на эту строку, обычная двубайтная переменная в памяти.

```
1                 .DSEG
2 StrPtr:         .data   2
```

Далее, в главной программе, в любом нужном нам месте, загружаем адрес текстовой строки в этот указатель, не забыв умножить его на два. Т.к. компилятор флеш адресует в словах, а контроллер оперирует только байтами:

```
1 Main:           NOP                             ; Любой произвольный код главной программы
2                 NOP
3                 NOP
4
5                 LDI     R17,low(2*String)       ; Берем младший байт
6                 LDI     R18,High(2*String)      ; Берем старший байт
7
8                 STS     StrPtr,R17              ; Сохраняем Младший байт
9                 STS     StrPtr+1,R18            ; Сохраняем Сташрий байт
```

И сразу же запускаем передачу, путем разрешения прерываний по UDRE. Так как UDR у нас пуст, то прерывание стартует мгновенно.

```
1                 LDI     R16, (1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(1<<UDRIE)
2                 OUT     UCSRB, R16
3
4 ;После чего спокойно выполняем любой другой код.
5                 NOP
6                 NOP
7                 NOP
```

А дальше все сделает наш обработчик прерывания события UDR Empty. На который ссылается вектор прерывания по опустошению регистра UDR:

```
1  UD_OK:          PUSHF                   ; Макрос, сохраняющий SREG и R16
2                  PUSH    ZL              ; Сохраняем в стеке Z
3                  PUSH    ZH
4
5                  LDS     ZL,StrPtr       ; Грузим указатели в индексные регистры
6                  LDS     ZH,StrPtr+1
7
8                  LPM     R16,Z+          ; Хватаем байт из флеша. Из нашей строки
9
10                 CPI     R16,0           ; Если он не ноль, значит читаем дальше
11                 BREQ    STOP_RX         ; Иначе останавливаем передачу
12
13                 OUT     UDR,R16         ; Выдача данных в усарт.
14
15                 STS     StrPtr,ZL       ; Сохраняем указатель
16                 STS     StrPtr+1,ZH     ; обратно, в память
17
18 Exit_RX:        POP     ZH              ; Все достаем из стека, выходим.
19                 POP     ZL
20                 POPF
21                 RETI
22
23 ; глушим прерывание по опустошению, выходим из обработчика
24 STOP_RX:        LDI     R16, (1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(0<<UDRIE)
25                 OUT     UCSRB, R16
26                 RJMP    Exit_RX
```

Все, по прерыванию обработчик сам выгребет данные из флеша и сам автоматом себя забанит когда дойдет до нуля в строке.

А зачем нужно прерывание TX_OK? Ну мало ли зачем. Еще какое-нибудь событие повесить. Оно сработает когда произойдет две вещи — UDR кончится и в сдвиговом регистре USART все биты улетят в провод. В принципе, многие юзают его вместо UDRE, но тогда между двумя уходящими байтами будет промежуток периодом в 1 байт. Что не очень кошерно.

Буферизация

Но далеко не всегда можно успеть обработать данные с большой скоростью. В этом случае их приходится куда-то складывать. А передавать/обрабатывать попозже. Особенно это касается отправки. Отправка медленная, а нам бы все сразу в буфер свалить и уйти по своим делам. А USART пускай там постепенно со всем этим разбирается.

Поэтому мы заведем два кольцевых буфера. На прием и на передачу (можно и только на передачу, это чаще требуется). А также ряд служебных переменных:

```
1  ; RAM =====================================================
2                  .DSEG
3
4                  .equ MAXBUFF_IN         =       10      ; Размер в байтах
5                  .equ MAXBUFF_OUT        =       10
6
7  IN_buff:        .byte   MAXBUFF_IN      ; Буфер приема
8  IN_PTR_S:       .byte   1               ; Указатель начала
9  IN_PTR_E:       .byte   1               ; Указатель конца
10 IN_FULL:        .byte   1               ; Флаг переполнения
11
12 OUT_buff:       .byte   MAXBUFF_OUT     ; Буфер передачи
13 OUT_PTR_S:      .byte   1               ; Указатель начала
14 OUT_PTR_E:      .byte   1               ; Указатель конца
15 OUT_FULL:       .byte   1               ; Флаг переполнения.
```

Указатели начала показывают откуда мы будем буфер читать, указатель конца нацелен на точку записи. В результате конец убегает, а начало его догоняет. Когда они сравняются — буфер пуст. При достижении максимальных границ буфера (10 байт) указатель переносится в начало, т.к. у нас буфер закольцован.

При этом может быть перехлест указателей между собой. Это даст ошибку переполнения и некорректную работу. Поэтому введены флаги переполнения, возникающие если конец пошел по второму кругу и догнал начало.

Сами указатели я сделал не классическими адресами, а смещениями относительно начала каждого буфера, Так получается оптимальней, не приходится сравнивать двубайтные числа. Но при этом длина буфера ограничена 255 байтами.

Прерывание по приему данных пишет в буфер IN_buff

```
1  RX_OK:          PUSHF                           ; Макрос, пихающий в стек SREG и R16
2                  PUSH    R17
3                  PUSH    R18
4                  PUSH    XL
5                  PUSH    XH
6
7                  LDI     XL,low(IN_buff)         ; Берем адрес начала буффера
8                  LDI     XH,high(IN_buff)
9                  LDS     R16,IN_PTR_E            ; Берем смещение точки записи
10                 LDS     R18,IN_PTR_S            ; Берем смещение точки чтения
11
12                 ADD     XL,R16                  ; Сложением адреса со смещением
13                 CLR     R17                     ; получаем адрес точки записи
14                 ADC     XH,R17
15
16                 IN      R17,UDR                 ; Забираем данные
17                 ST      X,R17                   ; сохраняем их в кольцо
18
19                 INC     R16                     ; Увеличиваем смещение
20
21                 CPI     R16,MAXBUFF_IN          ; Если достигли конца
22                 BRNE    NoEnd
23                 CLR     R16                     ; переставляем на начало
24
25 NoEnd:          CP      R16,R18                 ; Дошли до непрочитанных данных?
26                 BRNE    RX_OUT                  ; Если нет, то просто выходим
27
28
29 RX_FULL:        LDI     R18,1                   ; Если да, то буффер переполнен.
30                 STS     IN_FULL,R18             ; Записываем флаг наполненности
31
32 RX_OUT:         STS     IN_PTR_E,R16            ; Сохраняем смещение. Выходим
33
34                 POP     XH
35                 POP     XL
36                 POP     R18
37                 POP     R17
38                 POPF                            ; Достаем SREG и R16
39                 RETI
```

Если возникнет переполнение, то данные начнут затирать предыдущие, произойдет перехлест указателей, но перед этим поднимется флаг IN_FULL и диспетчер, или кто у нас там будет следить за всякими исключениями, может спешно среагировать и выгрузить буфер.

IN_FULL занимает целый байт, но можно было бы в один бит сделать. Я просто не стал усложнять. При чтении и записи в буфер в фоновой программе надо соблюдать атомарность, т.е. запрещать возможность записи в буфер из прерывания. А то будет трудно уловимый косяк.

Чтение из буфера приема делается тоже просто:

```
   ; Read from loop Buffer
1  ; IN: NONE
2  ; OUT:  R17 - Data,
3  ;              R19 - ERROR CODE
4
5  Buff_Pop:      CLI                             ; Запрещаем прерыания.
6                                                 ; Но лучше запретить прерывания конкретно
7  от
8                                                 ; UART, чем запрещать вообще все.
9                 LDI     XL,low(IN_buff)         ; Берем адрес начала буффера
10                LDI     XH,high(IN_buff)
11                LDS     R16,IN_PTR_E            ; Берем смещение точки записи
12                LDS     R18,IN_PTR_S            ; Берем смещение точки чтения
13                LDS     R19,IN_FULL             ; Берм флаг переполнения
14
15                CPI     R19,1                   ; Если буффер переполнен, то указатель
16 начала
17                BREQ    NeedPop                 ; Равен указателю конца. Это надо учесть.
18
19                CP      R18,R16                 ; Указатель чтения достиг указателя записи?
20                BRNE    NeedPop                 ; Нет! Буффер не пуст. Работаем дальше
21
22                LDI     R19,1                   ; Код ошибки - пустой буффер!
23
24                RJMP    _TX_OUT                 ; Выходим
25
26 NeedPop:       CLR     R17                     ; Получаем ноль
27                STS     IN_FULL,R17             ; Сбрасываем флаг переполнения
28
29                ADD     XL,R18                  ; Сложением адреса со смещением
30                ADC     XH,R17                  ; получаем адрес точки чтения
31
32                LD      R17,X                   ; Берем байт из буффера
33                CLR     R19                     ; Сброс кода ошибки
34
35                INC     R18                     ; Увеличиваем смещение указателя чтения
36
37                CPI     R18,MAXBUFF_OUT         ; Достигли конца кольца?
38                BRNE    _TX_OUT                 ; Нет?
39
40                CLR     R18                     ; Да? Сбрасываем, переставляя на 0
41
42 _TX_OUT:       STS     IN_PTR_S,R18            ; Сохраняем указатель
43                SEI                             ; Разрешаем прерывания
                  RET
```

Тут только один момент хитрый. Если у нас буфер переполнился, но не перехлестнулся, то его начало и конец совпадают, что как бы сигнализирует о том, что буфер пуст. Но на самом деле он полон непрочитанных данных и об этом сигнализирует флаг переполнения. Поэтому надо вначале проверять его, чтобы при не обходимости игнорировать признак равенства указателей.

На выходе функции Buff_Pop у нас в регистрах идут данные (R17) и код ошибки (R19). Если функция вернула 1, значит буфер пуст и регистр с данными можно игнорировать.

Запись в буфер задача более востребованная и чаще встречающаяся. И делается по аналогии с прерыванием RX

```
1  ; Load Loop Buffer
2  ; IN R19       - DATA
3  ; OUT R19      - ERROR CODE
4  Buff_Push:     CLI                             ; Запрет прерываний.
5                 LDI     XL,low(OUT_buff)        ; Берем адрес начала буффера
6                 LDI     XH,high(OUT_buff)
```

```
7                LDS     R16,OUT_PTR_E          ; Берем смещение точки записи
8                LDS     R18,OUT_PTR_S          ; Берем смещение точки чтения
9
10               ADD     XL,R16                 ; Сложением адреса со смещением
11               CLR     R17                    ; получаем адрес точки записи
12               ADC     XH,R17
13
14
15               ST      X,R19                  ; сохраняем их в кольцо
16               CLR     R19                    ; Очищаем R19, теперь там код ошибки
17                                              ; Который вернет подпрограмма
18
19               INC     R16                    ; Увеличиваем смещение
20
21               CPI     R16,MAXBUFF_OUT        ; Если достигли конца
22               BRNE    _NoEnd
23               CLR     R16                    ; переставляем на начало
24
25 _NoEnd:       CP      R16,R18                ; Дошли до непрочитанных данных?
26               BRNE    _RX_OUT                ; Если нет, то просто выходим
27
28
29 _RX_FULL:     LDI     R19,1                  ; Если да, то буффер переполнен.
30               STS     OUT_FULL,R19           ; Записываем флаг наполненности
31                                              ; В R19 остается 1 - код ошибки
32 переполнения
33
34 _RX_OUT:      STS     OUT_PTR_E,R16          ; Сохраняем смещение. Выходим
35               SEI                            ; Разрешение прерываний
                 RET
```

Пользоваться просто:
в R19 пихаем данные, вызваем функцию. НА выходе, в регистре R19 у нас код ошибки. Если там 1, то следующий байт писать нельзя — будет переполнение и перехлест указателей.

После загона данных в буфер надо запустить передачу. Я под это дело написал макрос

```
1        .MACRO  TX_RUN
2                LDI     R16, (1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(1<<UDRIE)
3                OUT     UCSRB, R16
4        .ENDM
```

Мы просто разрешаем прерывание UDRIE, а так как буфер пуст, то оно выполнится немедленно.

Из буфера данные забирает прерывание по опустошению UDR

```
1  UD_OK:        PUSHF
2                PUSH    R17
3                PUSH    R18
4                PUSH    R19
5                PUSH    XL
6                PUSH    XH
7
8
9                LDI     XL,low(OUT_buff)       ; Берем адрес начала буффера
10               LDI     XH,high(OUT_buff)
11               LDS     R16,OUT_PTR_E          ; Берем смещение точки записи
12               LDS     R18,OUT_PTR_S          ; Берем смещение точки чтения
13               LDS     R19,OUT_FULL           ; Берм флаг переполнения
14
15               CPI     R19,1                  ; Если буффер переполнен, то указатель
```

```
16 начала
17                BREQ    NeedSend                ; Равер указателю конца. Это надо учесть.
18
19                CP      R18,R16                 ; Указатель чтения достиг указателя записи?
20                BRNE    NeedSend                ; Нет! Буффер не пуст. Надо слать дальше
21
22                LDI     R16,1<<RXEN|1<<TXEN|1<<RXCIE|1<<TXCIE|0<<UDRIE       ; Запрет
23 прерывания
24                OUT     UCSRB, R16                                          ; По пустому UDR
25                RJMP    TX_OUT                  ; Выходим
26
27 NeedSend:      CLR     R17                     ; Получаем ноль
28                STS     OUT_FULL,R17            ; Сбрасываем флаг переполнения
29
30                ADD     XL,R18                  ; Сложением адреса со смещением
31                ADC     XH,R17                  ; получаем адрес точки чтения
32
33                LD      R17,X                   ; Берем байт из буффера
34                OUT     UDR,R17                 ; Отправляем его в USART
35
36                INC     R18                     ; Увеличиваем смещение указателя чтения
37
38                CPI     R18,MAXBUFF_OUT         ; Достигли конца кольца?
39                BRNE    TX_OUT                  ; Нет?
40
41                CLR     R18                     ; Да? Сбрасываем, переставляя на 0
42
43 TX_OUT:        STS     OUT_PTR_S,R18           ; Сохраняем указатель
44
45                POP     XH
46                POP     XL
47                POP     R19
48                POP     R18
49                POP     R17
                  POPF                            ; Выходим, достав все из стека
                  RETI
```

В Случае переполненного буфера мы сбрасываем флаг переполнения, а когда данные все выгребем и указатели вновь сравняются, то мы запрещаем прерывание по UDRE. Тем самым, остановив передачу.

И кратенький пример на работу с буфером. Ничего не показывает и не доказывает. Просто гоняет данные не напрямую, а через буфера:

```
1                 ...
2                 RCALL   Buff_Pop        ; Берем данные из буфера
3                 CPI     R19,1           ; Они там есть?
4                 BREQ    LOOPS           ; Нет? Ну еще раз
5
6                 INC     R17             ; Данные увеличили на 1, просто так.
7
8                 MOV     R19,R17         ; Переложили в R19
9  NewTry:        RCALL   Buff_Push       ; Пихнули в буфер
10
11                CPI     R19,1           ; Буфер не переполнился?
12                BRNE    RUN
13
14                TX_RUN                  ; Если да, то запускаем передачу
15                RCALL   Delay           ; Ждем или передаем управление другой задаче
16                RJMP    NewTry          ; А потом снова пробуем положить в буфер.
17
18 RUN:           TX_RUN                  ; Если все ок, запускаем передачу
19
```

20 . . .

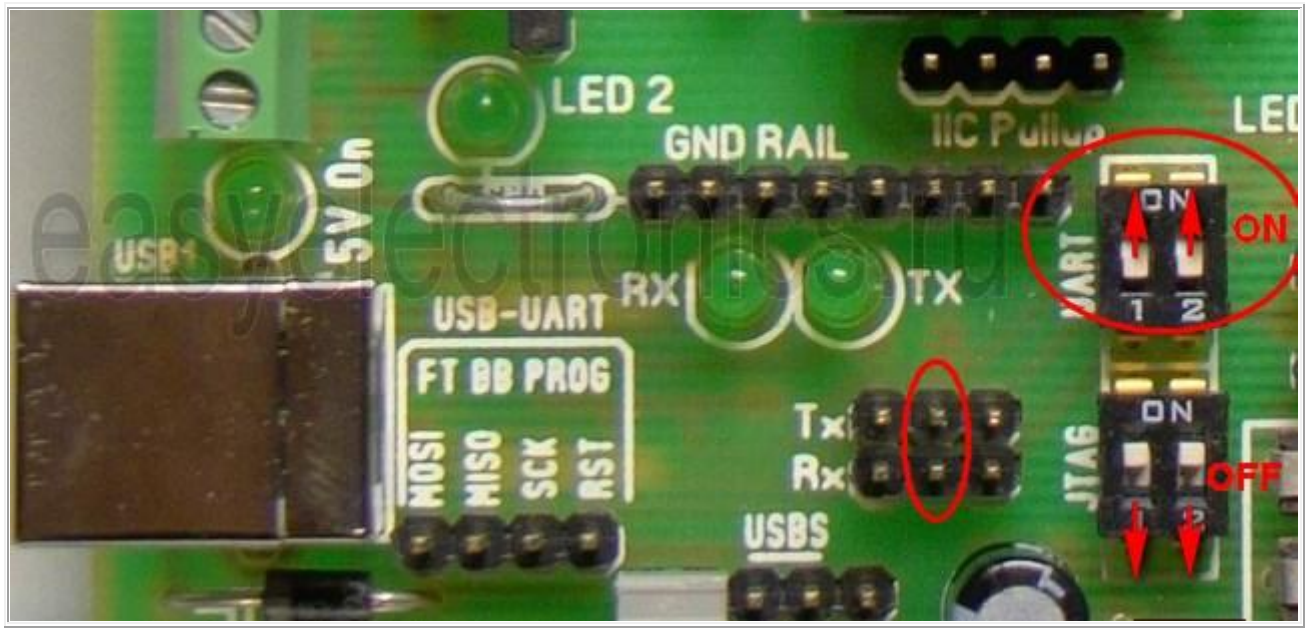
Примерно так.

Исходник с примером на буфере [3]

Бег по граблям
А теперь разберем ряд основных проблем с которыми можно столкнуться при освоении USART, а также с методами их решения.

Пожалуй самый распространенный баг это циклическая инициализация. Т.е. когда инициализация USART засунута в главный цикл и каждую итерацию происходит его переинициализация. Разумеется он от такого затраха офигевает и отказывается работать. Поэтому сразу запомните раз и навсегда — все инициализации делаются только один раз.
Нет, разумеется можно потом что-нибудь подправить и заново переинициализировать, но не тогда когда идет передача. Сначала пусть устройство завершит свои дела, а потом уж его можно трогать за регистры.

Второй популярный косяк — аппаратные проблемы. Т.е. пыжишься ты с кодом, перебираешь биты настроек, а он не работает. И код уже проверен на десяток раз, а все никак. А проблема вполне может быть и с зависшим COM портом или FTDI. Либо подключил что-то неправильно. Самый простой способ проверить интерфейс это отключить от контроллера передающую линию (на Pinboard [2] надо первести переключатель USART в положение OFF)



Замкнуть RX на TX на выходе FTDI (или MAX232). Подключиться терминальной программой, например Terminal v1.9b. И отправить в порт байт. Он должен вернуться обратно. Если возвращается что-то не то, значит проблема в интерфейсе.

Третьими граблями (а может даже и первыми) являются проблемы со скростями и тактовыми частотами. Ведь передача то асинхронная, а значит если скорость не та на которую мы запрограммированы, то коннекта у нас не выйдет. Самая засада в том, что скорость контроллера так сразу и не определишь. Чуть ошибся в фуз битах и запустил, например, контроллер не на 8мгц, а на 1мгц. Либо выставил тактовый генератор, да завелся он не на частоте кварца, а на какой нибудь из гармоник (редко, но бывает). А еще в некоторых контроллерах есть фуз бит CKDIV8 который делит тактовую частоту на 8. В общем, если UART не работает или работает с ошибками, то узнайте ТОЧНО на какой реально скорости у вас работает контроллер. Сделать это можно, например, с помощью таймера. Настроив его так, чтобы он мигал раз в секунду. Если мигает так как положено — значит частота верная. А если нет, то искать почему — фузы, биты делителя, кварцы и тды.

По этой же причине, кстати, если в готовом устройстве написано, что надо кварц на 12мгц, то ставить надо на 12мгц! Не на 10, не на 16 или 7.32, а именно на 12. Т.к. скорей всего от этого кварца зависят тайминги протоколов.

Четвертые грабли — бездумное копирование инициализации из всяких обучалок. Внимательно смотрите что у вас включено и на что настроено! Какие прерывания включены/выключены. Если прерывание есть, а обработчика нет, то вы получите неслабый глюк! Особенно это касается прерывания по RXC, которой пока из UDR не считаешь не успокоится.

Ну и в пятых — следите за совпадением скорости передатчика и приемника. Т.е. если передача идет на 9600, 1старт, один стоп, без четности. То и принимать надо ее на тех же параметрах. И никак иначе.

Извращенский ШИМ из UART

Пока писал статью про UART пришла в голову одна извращенная идея — на базе UART же можно организовать самый натуральный низкодискретный ШИМ!

Достаточно только сделать где-нибудь в памяти переменную, куда мы будем совать число с заданной скважностью нулей и единиц, а по прерыванию опустошения буфера это число снова пихать в регистр UDRE. Таким образом, генерация ШИМ будет самопроизвольной, без лишних телодвижений. Правда можно получить всего 10 разных значений ШИМ, но зато нахаляву!!!

Для тех кто не понял как, приведу числа которые надо будет непрерывно слать через UART:
два дополнительных значения мы получим за счет старт и стоп битов.

00000000 — 1/10
00000001 — 2/10
00000011 — 3/10
00000111 — 4/10
00001111 — 5/10
00011111 — 6/10
00111111 — 7/10
01111111 — 8/10
11111111 — 9/10

Да и частоты там можно получить нефиговые!
Красота!=)))))

AVR. Учебный Курс. Использование EEPROM

Иногда нужно сохранить данные так, чтобы они восстановились после перезагрузки контроллера. В этом тебе поможет EEPROM, почти все контроллеры серии AVR имеют на борту некоторое количество этой памяти. Физически и логически эта память находится в отдельном адресном пространстве, а чтение из EEPROM и запись туда осуществляется через специальные порты.

Чтобы что-то записать в **EEPROM** нужно в регистры адреса **EEARH** и **EEARL** (EEPROM Address Register) положить адрес ячейки в которую мы хотим записать байт. После чего нужно дождаться готовности памяти к записи – **EEPROM** довольно медленная штука. О готовности к записи нам доложит флаг **EEWE** (EEPROM Write Enable) регистра управления состоянием **EECR**, когда он будет равен 0, то память готова к следующей записи. Сам байт, который нужно записать, помещается в регистр **EEDR** (EEPROM Data Register). После чего взводится предохранительный бит **EEMWE** (EEPROM Master Write Enable), а затем, в течении четырех тактов, нужно установить бит **EEWE** и байт будет записан. Если в течении четырех тактов не успеешь выставить бит **EEWE** то предохранительный бит **EEMWE** сбросится и его придется выставлять снова. Это сделано для защиты от случайной записи в **EEPROM** память.

Чтение происходит примерно аналогичным образом, вначале ждем готовности памяти, потом заносим в регистры нужный адрес, а затем выставляем бит чтения **EERE** (EEPROM Read Enable) и следующей командой забираем из регистра данных **EEDR** наше число, сохраняя его в любом регистре общего назначения. Чтобы было понятно, я тебе набросал две процедурки – на чтение и на запись. Чтобы **записать байт**, нужно в регистры **R16** и **R17** занести **младший и старший байт адреса** нужной ячейки, а в регистр **R21** байт который мы хотим записать. После чего вызвать процедуру записи. Аналогично и с чтением – в регистра **R16** и **R17** адрес, а в регистре **R21** будет считанное значение.

Вот так выглядит запись в память:

```
1         …
2         LDI     R16,0           ; Загружаем адрес нулевой ячейки
3         LDI     R17,0           ; EEPROM
4         LDI     R21,45          ; и хотим записать в нее число 45
5         RCALL   EEWrite         ; вызываем процедуру записи.
```

А так чтение:

```
1         LDI     R16,0           ; Загружаем адрес нулевой ячейки
2         LDI     R17,0           ; EEPROM из которой хотим прочитать байт
3         RCALL   EERead          ; вызываем процедуру чтения. После которой
4                                 ; в R21 будет считанный байт.
```

Ну и, разумеется, сами процедуры чтения и записи

```
1   EEWrite:
2         SBIC    EECR,EEWE               ; Ждем готовности памяти к записи. Крутимся в
3   цикле
4         RJMP    EEWrite                 ; до тех пор пока не очистится флаг EEWE
5
6         CLI                             ; Затем запрещаем прерывания.
7         OUT     EEARL,R16               ; Загружаем адрес нужной ячейки
8         OUT     EEARH,R17               ; старший и младший байт адреса
9         OUT     EEDR,R21                ; и сами данные, которые нам нужно загрузить
10
11        SBI     EECR,EEMWE              ; взводим предохранитель
12        SBI     EECR,EEWE               ; записываем байт
13
14        SEI                             ; разрешаем прерывания
15        RET                             ; возврат из процедуры
16
17
18  EERead:
19        SBIC    EECR,EEWE               ; Ждем пока будет завершена прошлая запись.
20        RJMP    EERead                  ; также крутимся в цикле.
21        OUT     EEARL, R16              ; загружаем адрес нужной ячейки
22        OUT     EEARH, R17              ; его старшие и младшие байты
23        SBI     EECR,EERE               ; Выставляем бит чтения
24        IN      R21, EEDR               ; Забираем из регистра данных результат
          RET
```

Да, при работе с EEPROM нужно в цикле ожидания готовности не забывать командой WDR сбрасывать Watch Dog Timer — специальный сторожевой таймер, отслеживающий зависание процессора. Если его не сбрасывать с нужной периодичностью, то он сбрасывает контроллер. Это, конечно, если Watch Dog используется. По дефолту он вырублен. Но помнить надо, иначе огребете трудно отслеживаемый глюк.

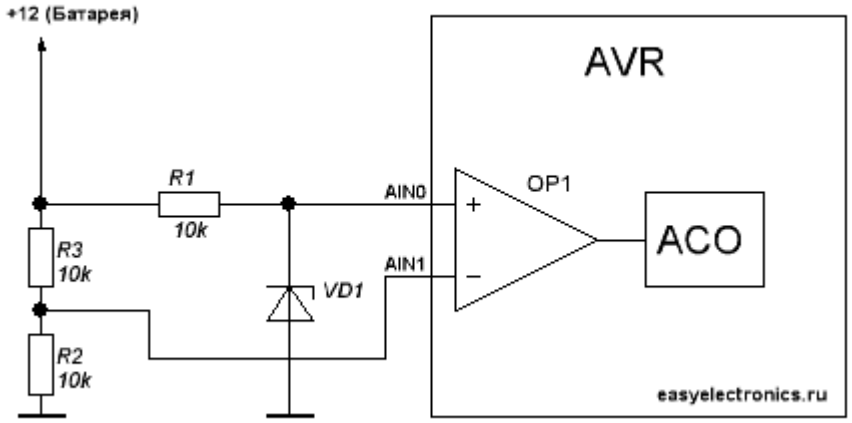
Впрочем, у EEPROM тоже есть свои прерывания. Это:

```
1          .ORG $01E
2          RETI                 ; (EE_RDY) EEPROM Ready
```

И никто не помешает выбросить цикл ожидания и сделать массовую запись в ЕЕПРОМ на прерываниях! Аналогично как это сделано для USART [1]. А если надо что то сохранить очень быстро, то можно и буферизированную с пробросом через RAM таким же образом запись заюзать. Т.е. сначала быстро сожрали в оперативку, а потом, неспеша, по прерываниям, загнать в EEPROM.

AVR. Учебный курс. Использование аналогового компаратора

Есть почти в каждой **AVR**ке, такая полезная приблуда как **аналоговый компаратор**. Это уже почти стандартное устройство и встречается очень часто во множестве разных контроллеров. Даже в древнем, как говно мамонта, *AT89C2051* он уже есть. Штучка прикольная, позволяет сравнивать два аналоговых сигнала и выносить свой вердикт 0 первый больше второго, 1 второй больше первого.



Применить его можно, например, чтобы отслеживать уровень заряда аккумулятора по просадке напряжения. Схема простейшая — стабилитрон создает опорное напряжение, которое всегда одиннаково, а напряжение с резистивного делителя зависит от входного напряжения.

Например, на входе у нас 8 вольт. Со стабилитрона, рассчитанного на 3.3 вольта, выходит всегда одно и то же напряжение — 3.3 вольта. А с симметричного резистивного делителя выходит половина напряжения то есть 4 вольта. 4 это больше чем 3.3, (3.3 — 4 = -0.7 результат меньше нуля) с компаратора выходит 0

Теперь если просядет батарейка и напряжение снизится до 6 вольт, то с делителя будет уже 3 вольта, а с опорного как было 3.3 так и осталось. Зато теперь на компараторе ситуация в корне поменялась — 3 меньше чем 3.3 (3.3 — 3 = 0.3 результат больше нуля) , а значит на выходе у него будет 1

Вот так, например, можно легко и просто следить за питающим напряжением и выдавать сигнал тревоги если батарейка сядет.

Настройка компаратора в контроллере AVR ATMega16
Для других моделей **AVR** все очень и очень похоже, просто мне так удобней. Если будет затруднение спросишь в комментах.
Мега16 имеет на борту компаратор, со входами **AIN0**(прямой вход) и **AIN1**(инверсный вход) Чтобы компаратор заработал его выводы нужно подключить на вход (**DDR**=0) и отключить подтяжку до единицы (**PORT**=0).

Регистр конфигураций компаратора ACSR
Биты:

- **ACD** включение компаратора 0 включен, 1 выключен. По дефолту там ноль, а значит при старте компаратор включен.
- **ACBG** - подключение к прямому входу компаратора внутреннего источника опорного напряжения (**ИОН**) на 1.22+(-)0.05V. Если 0 то **ИОН** не подключен.
- **ACO** — бит результата. Собственно, это и есть выход компаратора.
- **ACI** — флаг прерывания. Я думаю, что ты уже привык к тому, что в **AVR** есть прерывание на каждый чих. Компаратор не исключение. Устанавливается по событию, сбрасывается после ухода на обработчик либо программно, как всегда, записью в него 1.
- **ACIE** - где есть прерывание там должен быть и бит разрешения. Это он и есть. Установив в 1 мы разрешаем прерывания от компаратора. По дефолту, естественно, нуль.
- **ACIC** — подключение компаратора к **схеме захвата таймера1**. При попадании сигнала на схему захвата текущее значение с таймера тут же тырится в специальный регистр захвата, а таймер продолжает считать дальше. А в привязке к компаратору это удобно когда нужно измерять длительности сигналов.

- Биты **ACIS1:ACIS0** определяет условие возникновения прерывания от компаратора:
  - **00** — любое изменение на входе.
  - **01** - зарезервировано для следующих поколений
  - **10** — переход с 1 на 0
  - **11** — переход с 0 на 1

Вот за что люблю АТМеги так это за фарш! Даже свой собственный источник опорного напряжения есть. Так что из схемы со стабилитроном можно смело выкинуть все, что касается стабилитрона :) Оставив только делитель, ну и подобрав плечи резисторного делителя так, чтобы получить напругу на выходе чуть выше чем 1.22V.
Мало того, если в контроллере есть **АЦП,** то на вывод **AIN1** можно подключить **ЛЮБОЙ** вход АЦП. Нереально круто! Для того, чтобы это сделать нужно:

- В регистре **SFIOR** (регистр специальных функций) выставить бит **ACME** (вроде бы так ракеты назывались в мультике про койота и страуса ;))
- Выключить **АЦП**, сбросив бит **ADEN** в регистре настроек АЦП (**ADCSRA**)
- В регистре **ADMUX** в разрядах **MUX2:MUX1:MUX0** указать номер входа АЦП.

Пример:
В качестве примера я возьму своего робота. Сварганив ему систему контроля за питанием. Напряжение с аккумулятора проходит через делитель **1.5:10** в результате на 12 вольтах у нас на выходе будет **(1.5/(1.5+10))*12=1.56** и только при просадке напруги до 9.5 вольт на выходе делителя будет ниже 1.22 вольта. Заюзаю пока неиспользованный вход **PC5** (это вход **ADC5**)
Заправлю его через **АЦП** в компаратор, а в качестве опорного напряжения возьму встроенный **ION** который посажу на **AIN0** изнутри. При просадке напряжения буду зажигать контрольную лампу.

Что нам нужно:
Для начала подключить **ИОН** — бит **ACBG**
Далее, бит прерывания — в обработчике прерываний я включу лампочку
Ну и выставить условие по которому будет прерывание с 0 на 1. Дело в том, что на плюс-вход компаратора идет 1.22 с **ИОН**, а на минус-вход пойдет 1.5 с делителя. В результате **1.22-1.5<0**, следовательно на выходе 0. Ну, а как только ситуация поменяется будет 1.

```
1  ; Инициализирую компаратор
2          LDI     R16,(1<<ACBG)|(1<<ACIE)|(1<<ACIS1)|(1<<ACIS0)
3          OUT     ACSR,R16        ; Забрасываем в регистр
4
5          IN      R16,SFIOR       ; Достаем SFIOR
6          ORI     R16,(1<<ACME)   ; Выставялем в нем бит ACME
7          OUT     SFIOR,R16
8
9  ; АЦП у меня по дефолту вырублен, поэтому пока не напрягаюсь с его отключением
10         LDI     R16,5           ; подаю напругу на 5й вход АЦП
11         OUT     ADMUX,R16
```

Ну и собственно все, осталось только добавить в программу переход по вектору

```
1         .ORG    ACIaddr         ; Analog Comparator
2         RJMP    Battary_LOW
3
4 ;И добавить где нибудь в программе обработчик прерывания
5 Battary_LOW:    LIGHT_ON        ; Вызов макроса зажжения огня.
6                 RETI            ; Выход из прерывания.
```

Или вот, второй пример. Попроще.

На компаратор подается два напряжения. Одно опорное, со стабилизатора 3.3 вольта демоплаты Pinboard [1].
Второе с потенциометра, то которое мы сравниваем. Подаются напрямую на выводы компаратора AIN0 и AIN1
Когда напряжение с потенциометра выше чем 3.3 вольта с стабилизатора, то в компараторе у нас на выходе 0 (бит ACO=0). И наоборот. Также настроено прерывание так, чтобы при попадании в него происходила инверсия бита PD4 — на нем висит светодиод LED1

Код
Обработчик прерывания:

```
1 COMP_OK:        PUSHF                           ; Сохранили флаги
2                 PUSH    R17                     ; Сохранили R17
3
4                 INVB    PORTD,4,R17,R16         ; Инвертировали бит PD4
5
6                 POP     R17                     ; Достали все из стека
7                 POPF                            ; И флаги тоже
8
9                 RETI                            ; Вышли
```

Инициализация периферии:

```
1  ; Internal Hardware Init =====================================
2
3  ;Comparator Init
4                 OUTI    ACSR,1<<ACIE|1<<ACIS1|1<<ACIS0 ; Разрешаем прерывание
5                                                        ; Ловим 0-1 переход.
6  ; Ports Init
7                 SBI     DDRD,4
8                 SBI     DDRD,7
9
10                SEI
11 ; End Internal Hardware Init ================================
```

Главная программа:

```
1  ; Main ======================================================
2  Main:          IN      R16,ACSR        ; Взяли состояние компаратора
3                 ANDI    R16,1<<ACO      ; Оставили только бит состояния
4
5                 BST     R16,ACO         ; Скопировали его в T
6                 BLD     R16,7           ; Выгрузили в 7й разряд R16
7
8                 IN      R17,PORTD       ; Взяли значение из PORTD
9                 ANDI    R17,0b01111111  ; Подавили значение 7го разряда
10
11                OR      R17,R16         ; Наложили на него значение ACO
12
13                OUT     PORTD,R17       ; Вернули значение в порт.
14
15                RJMP    Main
```

Светодиод LED3 показывает состояние бита ACO, а LED1 фиксирует заходы в обработчик прерывания компаратора. При этом наблюдается мерзкий эффект — когда сравниваемые напряжения на входах компаратора очень близки, то возникает дребезг. Т.е. мельчайшие помехи уже начинают играть роль и перевешивают чашу весов компаратора то в одну то в другую сторону. Возникает жуткий дребезг. Этот дребезг надо подавлять программно. Скажем игнорировать изменения сигнала если он чаще чем раз в несколько миллисекунд.

На видео дребезг очень хорошо видно на экране осциллографа.

AVR. Учебный курс. Использование АЦП

Многие **AVR** имеют на борту **АЦП** последовательного приближения.
АЦП это десятиразрядное, но при точности +/- 2 минимально значащих разрядов его можно смело считать восьмиразрядным :) Так как в младших двух разрядах всегда мусор какой то, не похожий на полезный сигнал.

Тем не менее это неплохой инструмент для контроля напряжения, в восьмиразрядном режиме имеющий **256 отсчетов** и выдающее частоту дискретизации **до 15кГц** (15 тысяч выборок в секунду).

Конфигурация источника
Сигнал в **АЦП** подается через мультиплексор, с одного из восьми (в лучшем случае, часто бывает меньше) входов. Выбор входа осуществляется регистром **ADMUX**, а точнее его битами **MUX3...MUX0**. Записанное туда число определяет выбраный вход. Например, если **MUX3..0** = **0100**, то подключен вывод **ADC4**.

Кроме того, существует несколько служебных комбинаций битов **MUX**, использующихся для калибровки. Например, **1110** подключает к **АЦП внутренний источник опорного напряжения на 1.22 вольта**. А если записать в **MUX3..0** все единицы, то **АЦП** будет изнутри посажено на землю. Это полезно для выявления разных шумов и помех.

У старших **AVR** семейства **Mega** (8535, 16, 32, 128) есть возможность включить **АЦП** в режиме **дифференциального входа**. Это когда на два входа приходят разные напряжения. Одно вычитается из другого, да еще может умножаться на коэффициент усиления. Зачем это нужно? А, например, когда надо замерить перекос напряжения измерительного моста. У какого-нибудь тензомоста при входном напряжении в пять вольт выходные сигналы будут различаться между собой всего лишь 30мВ, вот и поймай его. А так подал на диф вход, подогнал нужный коэффициент усиления и красота!

Таблицу значений **MUX3..0** для диф включения я не буду тут приводить, она находится легко в даташите, зовется она «**Input Channel and Gain Selections**«. Я поясню лишь один тонкий момент. В режиме выбора диф входа встречаются такие комбинации как: **первый вход ADC0 и второй вход тоже ADC0** ну и коэффициент усиления еще. Как так? Ведь для диф входа нужно **два разных** входа! Вначале подумал опечатка, поднял даташит на другую АВРку — та же ботва. Потом повтыкал в текст ниже и понял — это для калибровки нуля. Т.е. перед началом съема диф данных нам нужно закоротить входы, чтобы понять, что же у нас ноль. Так вот, комбинация когда два входа подключены к одной ноге это и есть та самая калибровочная закоротка входов. Делаешь первое преобразование на такой фигне, получаешь **смещение нуля**. А потом вычитаешь его из всех полученных значений, что резко повышает точность.

Мультиплексирование каналов осуществляется только после того, как завершится преобразование, поэтому можно смело запускать **АЦП** на обсчет входных значений, записывать в MUX3..0 параметры другого входа, и готовится снимать данные уже оттуда.

Выбор опорного сигнала
Это максимальное напряжение, которое будет взято за максимум при измерениях. Опорное напряжение должно быть как можно стабильней, без помех и колебаний — от этого кардинальным образом зависит точность работы **АЦП**. Задается он в битах **REFS1..0** регистра **ADMUX**.

- По дефолту там стоит **REFS1..0 = 00** — внешний **ИОН**, подключенный к входу **AREF**. Это может быть напряжение со специальной микросхемы опорного напряжения, или же со стабилитрона какого, если нужно замерять небольшое напряжение, заметно меньшее чем напряжение питания, скажем от 0 до 1 вольт, то чтобы было точнее, и чтобы оно не затерялось на фоне пятивольтового питания, то на AREF мы заводим опорное напряжение в 1 вольт.
- **REFS1..0 = 01** — тут просто берется напряжение питания. У всех почти Мег с **АЦП** есть вход **AVCC** — вот это напряжение питания для **АЦП** и порта на который это **АЦП** повешено. Подавать туда плюс питания желательно через **LC** фильтр, чтобы не было искажений.
- **REFS1..0 = 11** — внутренний источник опорного напряжения **на 2.56 вольт**. Честно говоря, качество этого источника мне сильно не понравилось. С ним показания **АЦП** плавают как говно в проруби. Но если невозможно обеспечить гладкую и стабильную подачу напряжения на **AREF** или **AVCC** то прокатит. Кстати, внутренний **ИОН** подключен к выводу **AREF** так что можно повесить туда кондер и попробовать его чуть чуть сгладить. Немного, но помогает.

Выбор режима запуска преобразования
В регистре **SFIOR** под **АЦП** отведено аж три бита. **ADTS2..0** которые управляют режимами запуска **АЦП**.

- По дефолту **ADTS2..0 = 000** и это значит, что преобразование идет в непрерывном режиме. Ну или по ручному запуску.
- **ADTS2..0 = 001** — запуск **АЦП** от аналогового компаратора. Удобно блин. Например, чтобы не замерять постоянно входную величину, а запрограммировать компаратор на то, что как только у него вылезет что-либо выше порога, так тут же захватывать это дело на **АЦП**.
- **ADTS2..0 = 010** — запуск от внешнего прерывания **INT0**

- **ADTS2..0 = 011** — по совпадению таймера **T0**
- **ADTS2..0 = 100** — по переполнению таймера **T0**
- **ADTS2..0 = 101** — по совпадению с таймера **T1**
- **ADTS2..0 = 110** — По переполнению таймера **T1**
- **ADTS2..0 = 111** — По событию «захват» таймера **T1**

Скорость работы АЦП
Частота выборки **АЦП** задается в битах предделителя **ADPS2...0** регистра **ADCSR**. Саму таблицу можно поглядеть в даташите на соответствующий МК, скажу лишь то, что самая оптимальная точность работы модуля **АЦП** находится в пределах **50...200кГц**, поэтому предделитель стоит настраивать исходя из этих соображений. С повышением частоты точность падает.

Прерывания.
Естественно у **АЦП** есть прерывания. В данном случае это прерывание по окончанию преобразования. Его можно разрешить битом **ADIE**, а внаглую вручную палится оно по флагу **ADIF** (регистр **ADCSRA**). Флаг **ADIF** автоматом снимается при уходе на вектор прерывания по **АЦП**.

Данные с **АЦП** сваливаются в регистровую пару **ADCH:ADCL** откуда их можно забрать. Причем тут есть один прикольный момент. Регистровая пара то у нас ведь 16ти разрядная, а **АЦП** имеет разрядность **10бит**. В итоге, лишь один регистр занят полностью, а второй занимает лишь оставшиеся два бита. Так вот, выравнивание может быть как по правому краю — старшие два бита в **ADCH**, а младшие в **ADCL**, либо по левому — старшие биты в **ADCH**, а два младших бита в **ADCL**.

[x][x][x][x][x][x]**[9][8]:[7][6][5][4][3][2][1][0]** или **[9][8][7][6][5][4][3][2]:[1][0]**[x][x][x][x][x][x]

Зачем это сделано? А это **выборка разрядности** так оригинально организована. Как я уже говорил, в младших разрядах все равно мусор и шумы (по крайней мере я от них так и не смог избавиться, как ни старался) . Так вот. Делаем выравнивание по левому краю. И загребаем старшие разряды **только из регистра ADCH**, а на младший забиваем. Итого, у нас число отсчетов становится 256. За выравнивание отвечает бит **ADLAR** в регистре **ADMUX** 0 — выравнивание по правой границе, 1 — по левой.

Запуск преобразования в ручном или непрерывном режиме.
Для запуска преобразования нужно вначале разрешить работу **ADC**, установкой бита **ADEN** в регистре **ADCSR** и в том же регистре ткнуть в бит **ADSC**. Для запуска непрерывного преобразование (одно за другим) нужно также выставить бит **ADFR** (**ADATE** в некоторых **AVR**).

Повышение точности уходом в спячку.
Для повышения точности, чтобы внутренние цепи **АЦП** не гадили своими шумами, можно запустить **АЦП** в **спящем режиме**. Т.е. проц останавливается, все замирает. Работает только **WatchDog** и блок **АЦП**. Как только данные сосчитаются, генерируется прерывание которое будит процессор, он уходит на обработчик прерывания от **АЦП** и дальше все своим чередом.

А теперь приведу парочку примеров простой инициализации и работы с **АЦП**. Микроконтроллер **ATMega16**

```
1  ; Мой любимый макрос для записи в порты :))))
2          .MACRO outi
3          LDI     R16,@1
4          OUT     @0,R16
5          .ENDM
6
7  ; ADC Init - Инициализурем АЦП. Это можно сунуть куда - нибудь в начало кода
8
9          OUTI    ADCSRA,(1<<ADEN)|(1<<ADIE)|(1<<ADSC)|(1<<ADATE)|(3<<ADPS0)
10         ; Итак что тут у нас:
11         ; ADEN = 1 - разрешаем АЦП
12         ; ADIE = 1Разрешаем прерывания.
13         ; ADSC = 1 Запускаем преобразование (первое, дальше автоматом)
14         ; ADATE = 1 Непрерывные последовательные преобразования, одно за другим.
15         ; ADPS2..0 = 3 Делитель частоты на 8 - так у меня получается оптимальная частота.
16
17         OUTI    ADMUX,0b01000101
18         ;А тут выбираем откуда брать будем сигнал
```

```
19          ;REFS -- 0b[01]000101 первые два бита - напряжение с входа AVCC
20          ;ADLAR --0b01[0]00101следующий бит выравнивание по правому краю
21          ;MUX -- 0b010[00000]</B>Сигнал на вход идет с нулевого канала АЦП.
```

А что дальше делать? А ничего! Сидеть и ждать прерывания!
Когда оно придет процессор кинет на вектор и дальше уже можно либо переписать данные из **ADCH:ADCL** в другое место, либо какую простенькую обработку тут же, не отходя от кассы, замутить. Вроде усреднения.

Вариант два, с уходом в спячку. В принципе, все то же самое, только нужно выключить автоматический перезапуск конвертирования. Далее в регистре **MCUCR** в битах **SM2..0** выбрать режим **ADC Noise Reduction SM2..0 = 001**, а после, сразу же после запуска послать процессор в спячку командой **SLEEP**. Как только он уснет заработает **АЦП,** сделает преобразование и проснется на прерывании.

Выглядит это так:

```
1  ; ADC Init - Инициализурем АЦП. Это можно сунуть куда - нибудь в начало кода
2
3          OUTI    ADMUX,0b01000101
4          ;А тут выбираем откуда брать будем сигнал
5          ;REFS -- 0b[01]000101 первые два бита - напряжение с входа AVCC
6          ;ADLAR --0b01[0]00101следующий бит выравнивание по правому краю
7          ;MUX -- 0b010[00101]</B>Сигнал на вход идет с 5й ноги.
8
9          OUTI    MCUCR,0b10010000
10         ; Выставил биты спящего режима в Noise Reduction
11
12
13 ; А это уже тело главной программы
14 Main Prog:
15         OUTI    ADCSRA,(1<<ADEN)|(1<<ADIE)|(1<<ADSC)|(0<<ADATE)|(3<<ADPS0)
16         ; Подготовили АЦП
17         SLEEP           ; Свалились в спячку, как только Хрюша со Степашей скажут нам
18                         ; Баю-бай так врубится молотилка АЦП, затем придет Фреди Крюг...
19                         ; в смысле запрос на прерывание и проц резко проснется
```

Ну и, для повышения точности, следует соблюдать ряд правил по подключению питания к **АЦП** модулю, например подавать напряжение на вход **AVCC** через дроссель, ставить конденсаторы и земли побольше вокруг. Об этом все есть в даташите. Я же скоро выложу пример рабочей программы — примера для **АЦП** и **UART**.

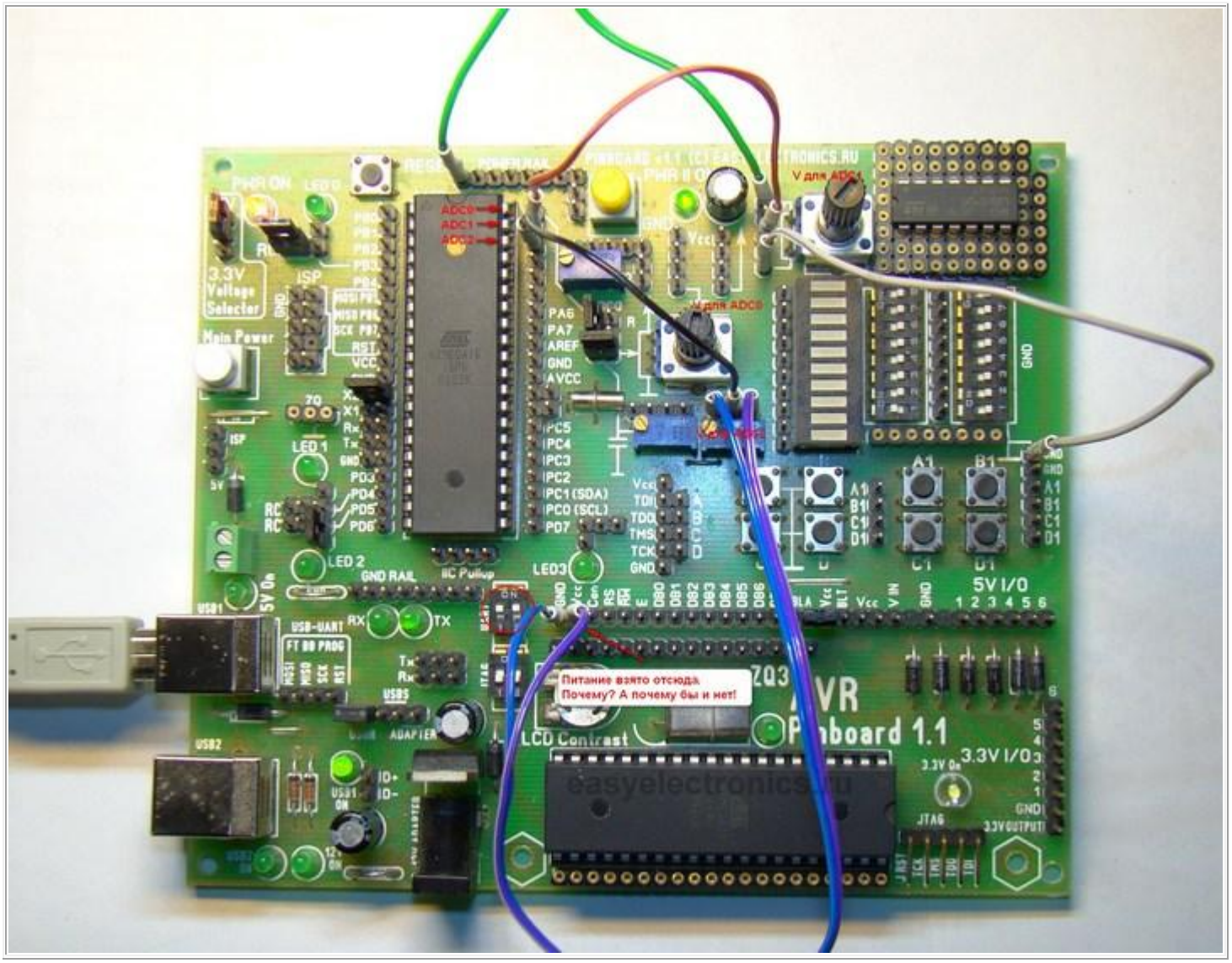
AVR. Учебный Курс. Выдача данных с АЦП на UART. Мультиплексирование каналов АЦП

Несколько постов назад я заикнулся о том, что выдам на гора программу-пример **для работы с АЦП**. Пора за базар отвечать :) Делать мы будем простенький цифровой вольтметр с замашками осциллографа. Точнее осциллографом это можно назвать с большой натяжкой, скорей **самописец**. Так, побаловаться.

Задача:
Получить по очереди напряжение с трех каналов **АЦП** и отправить его по последовательному порту в комп. По запросу с компа показвыать напряжение на каждом из измеряемых каналов. В компе полученный поток байт представить в виде графика.

Теоретическую часть [1] я уже разобрал, осталось поставить эксперимент в реальном железе.

Собираем схему на демоплате Pinboard [2].
Нам нужны три разных напряжения. Их проще всего получить с переменных резисторов, включенных потенциометрами. При этом средняя точка переменного резистора подключается к каналу АЦП, а крайние точки к +5 и GND питания. При этом при вращении рукоятки резистора напряжение на его средней точке будет меняться от нуля до +5 вольт. Резистор, подключенный к каналу ADC0 уже так включен и никаких лишних движений не требует. А вот два других надо будет подключить. На видео и на фотках хорошо видно что и куда идет.



Поглядеть крупнее [3]

Теперь осталось написать код. Код я сделаю на прерываниях и конечных автоматах.

Вначале переменные:

```
; RAM =====================================================
                .DSEG
RX_sel:         .byte 1         ; Переменная состояния отправки данных
ADCH_sel:       .byte 1         ; Переменная текущего канала АЦП
ADCCH:          .byte 8         ; Восемь байт под хранение результатов АЦП. По байту на
канал.
```

Таблица прерываний стандартная, ее я приводить не буду. Сами увидите в проекте, что можно будет скачать в конце статьи

Прерывание по приему байта. Работает просто. Принятый байт проверяет на код цифры 0, 1 или 2. Если совпадает — то выставляет соответствующую переменную и в терминал начинают валиться данные с этого канала АЦП. Если не совпадает — выставляет значение которое блокирует отправку любых данных.

```
; Interrupts ==============================================
RX_OK:          PUSHF                   ; Сохраняем SREG и R16 в стеке
```

```
3                IN      R16,UDR         ; Берем принятый байт
4
5                CPI     R16,'0'         ; Case 0
6                BREQ    Ch_0
7                CPI     R16,'1'         ; Case 1
8                BREQ    Ch_1
9                CPI     R16,'2'         ; Case 2
10               BREQ    Ch_2
11
12               LDI     R16,3           ; Defaul
13
14               RJMP    EXIT_RX
15
16 Ch_0:         LDI     R16,0           ; Если выбран 0 канал
17               RJMP    EXIT_RX
18
19 Ch_1:         LDI     R16,1           ; Если выбран 1 канал
20               RJMP    EXIT_RX
21
22 Ch_2:         LDI     R16,2
23               RJMP    EXIT_RX         ; Если выбран 2 канал
24
25
26 EXIT_RX:      OUT     UDR,R16         ; Даем эхо и инициируем передачу.
27               STS     RX_sel,R16      ; Сохраняем выбраное значение канала
28               POPF                    ; Достаем регистры из стека
29               RETI
```

Прерывание по окончании отправки одного байта берет следующее значение и шлет его снова. Тем самым инициализируюя непрерывную передачу данных по прерываниям.

```
1  ;---------------------------------------------------------------------
2  TX_OK:        PUSHF                   ; Сохраняем флаги и R16
3
4                LDS     R16,RX_sel      ; Смотрим какой канал надо слать
5
6                CPI     R16,0           ; Case 0
7                BREQ    Send0
8
9                CPI     R16,1           ; Case 1
10               BREQ    Send1
11
12               CPI     R16,2           ; Case 2
13               BREQ    Send2
14
15                                       ; Default
16               RJMP    EXIT_TX
17
18 Send0:        LDS     R16,ADCCH       ; Шлем выбраный канал в UART
19               OUT     UDR,R16         ; Смещение адреса канала задается относительно
20 базы ADCCH
21               RJMP    EXIT_TX
22
23 Send1:        LDS     R16,ADCCH+1     ; Тут лежат данные первого канала
24               OUT     UDR,R16
25               RJMP    EXIT_TX
26
27 Send2:        LDS     R16,ADCCH+2     ; А тут второго. Адреса все фиксированные.
28               OUT     UDR,R16
29               RJMP    EXIT_TX
30
31 EXIT_TX:      POPF                    ; Достаем все из стека, выходим.
                 RETI
```

Прерывание готовности АЦП работает несколько сложней. Суть в том, что работа АЦП у нас идет по такому алгоритму:
Запускаем одиночное АЦП преобразование и ждем прерывания готовности АЦП.
В прерывании забираем данные.
Сохраняем данные.
Переключаем канал с которого мы снимали показания.
Запускаем следующее преобразование (уже для другого канала)
Выходим из прерывания и ждем следюущего прерывания.

Данный алгоритм хорош тем, что все делается автоматически, в режиме конечного автомата. В результате у нас в памяти, в массиве ADCCH, всегда лежат 8 свежих значений, снятых с 8ми каналов АЦП. Остается их только считать и использовать. При этом главный цикл крутится по своим делам и не парится, зная, что свежие значение всегда его ждут.

В самом прерывании активно используется работа с масками. Для того, чтобы сменить номер канала. Номер канала лежит в последних трех битах регистра ADMUX и может иметь значение от 0 до 7 (000 и 111 соотвествтенно). И нам надо в каждом вызове увеличивать значение канала, перебирая их по очереди. Просто так икрементировать ADMUX нельзя, т.к. кроме номера канала там лежат еще и быти управления АЦП, выравнивания и опорного напряжения — они могут сбиться. Приходится изощряться с масками, чтобы их не задеть.

```
1  ADC_OK: PUSHF
2                  PUSH    ZL
3                  PUSH    ZH
4                  PUSH    R17
5
6                  IN      R16,ADMUX       ; Берем ADMUX
7                  ANDI    R16,0x07        ; Маской отрезаем лишние биты. Получаем номер
8  канала
9                                          ; С которого было снято измерение.
10
11                 MOV     R17,R16         ; Сохранили копию номера канала
12
13                 LDI     ZL,low(ADCCH)   ; Берем адрес начала массива с будущими данными.
14                 LDI     ZH,High(ADCCH)
15
16                 ADD     ZL,R16          ; Прибавляем к адресу наш номер канала.
17                                         ; Если было переполнение, то будет флаг C
18                 CLR     R16             ; Флаг важен, а значение в R16 уже нет. Но нам
19 нужен ноль
20                                         ; Возьмем и сделаем его из R16.
21                 ADC     ZH,R16          ; Сложим флаг возможного переполнения с ZH
22                                         ; Т.о. у нас получается в Z = адрес (ADCCH+номер
23 канала)
24                                         ; И значения из разных каналов ложатся в разные
25 переменные массива
26                                         ; с адресом базы ADCCH
27
28                 IN      R16,ADCL        ; Младшее значение нам не надо. Но считать его
29 нужно.
30                 IN      R16,ADCH        ; Берем в R16 значение из АЦП
31                 ST      Z,R16           ; Сохраняем его в массив по нужному адресу
32
33
34                 IN      R16,ADMUX       ; Опять взяли ADMUX
35                 ANDI    R16,0xF8        ; На этот раз обнулили номер канала. Оставив
36 остальные биты нетронутыми
37
38                 INC     R17             ; Увеличили на 1 заныченный ранее номер канала
39                 ANDI    R17,0x07        ; Обрезали лишние биты, чтобы не было
40 переполнения.
41                                         ; Число по маске 0x07 в принципе не может быть
42 больше 7.
43                                         ; А каналов у нас от 0 до 7. То что надо.
```

```
44
45                OR      R16,R17         ; Слепили старое значение из ADMUX с новым
46 значением номера
47                OUT     ADMUX,R16       ; Канала. Т.е. по факту сделали MUX = MUX+1 выбрав
48 следующий канал
                                          ; Спихнули его в регистр ADMUX. Все, следующий
   канал выбран. Можно запускать
                                          ; Следующее преобразование.

                  OUTI    ADCSRA,(1<<ADEN)|(1<<ADIE)|(1<<ADSC)|(0<<ADATE)|(3<<ADPS0)    ;
   Запустили

                  POP     R17             ; Корректный выход из прерывания.
                  POP     ZH
                  POP     ZL
                  POPF
                  RETI
```

Также в программе есть инициализация UART (я запустил его на скорости 1200) и инициализация и первичный запуск АЦП.
Стоит обратить внимание на инициализацию UART — там мы разрешаем прерывания по приему и передаче.

```
1  ;Internal Hardware Init =======================================
2
3  Reset:          STACKINIT                                       ; Инициализация стека
4                  RAMFLUSH                                        ; Очистка памяти
5
6  ; Usart INIT
7                  .equ    XTAL = 8000000
8                  .equ    baudrate = 1200
9                  .equ    bauddivider = XTAL/(16*baudrate)-1
10
11 uart_init:      LDI     R16, low(bauddivider)
12                 OUT     UBRRL,R16
13                 LDI     R16, high(bauddivider)
14                 OUT     UBRRH,R16
15
16                 LDI     R16,0
17                 OUT     UCSRA, R16
18
19 ; Прерывания разерешены, прием-передача разрешен.
20                 LDI     R16, (1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(0<<UDRIE)
21                 OUT     UCSRB, R16
22
23 ; Формат кадра - 8 бит, пишем в регистр UCSRC, за это отвечает бит селектор
24                 LDI     R16, (1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1)
25                 OUT     UCSRC, R16
26
27
28 ; ADC Init
29
30                 ; Опорное напряжение с AVCC, выравнивание влево, канал 0.
31                 OUTI    ADMUX,1<<REFS0|1<<ADLAR|0<<MUX0
32
33 ; End Internal Hardware Init ================================
34
35
36 ; Run ===================================================
37 ; Запуск однократного преобразования. Все остальное сделаем в прерываниях.
38
39                 SEI
40                 OUTI    ADCSRA,(1<<ADEN)|(1<<ADIE)|(1<<ADSC)|(0<<ADATE)|(3<<ADPS0)
41
```

```
42 ; End Run ============================================================
```

Первый раз АЦП запускается вручную. А потом начинает молотить по прерываниям самостоятельно.

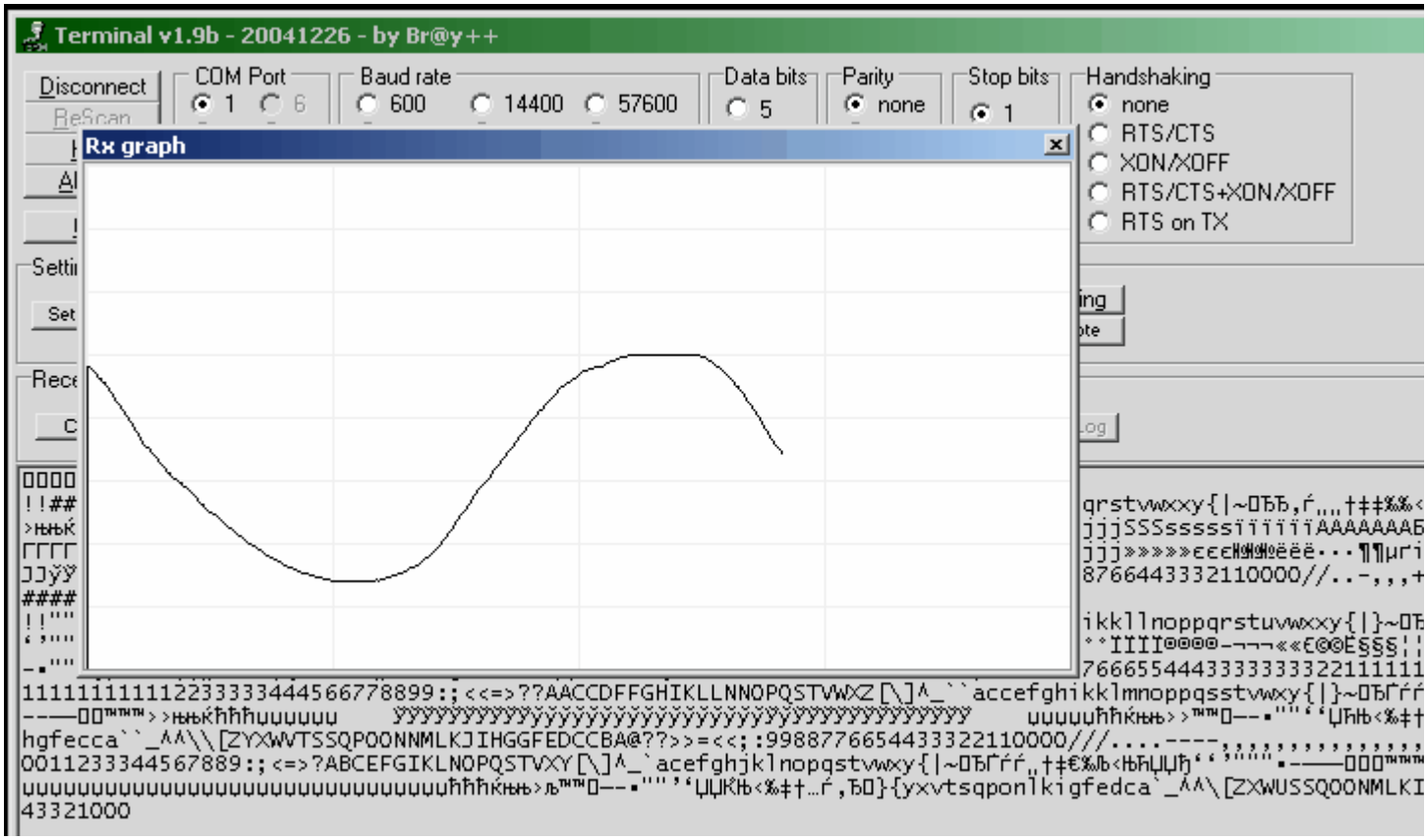
Главный же цикл вообще пустой. Он тут и не нужен. Там можно смело писать любой код, а данные АЦП загребая из памяти. Главное помнить где они лежат ;)

```
1 ; Main ================================================================
2 Main:           NOP     ; Весь цимес в прерываниях. Нам тут делать нефиг.
3
4                 RJMP    Main
```

Прошиваем программу в контроллер, жмем RESET

Запускаем теперь чумовую программку **Terminal** [4] и подсоединяемся на скорости 1200бод. Ничего не происходит, но не беда. Надо сказать контроллеру какой какой вывод АЦП мы хотим поглядеть. Отправляем 0 — в ответ нам полетят данные с нулевого канала. Если отправить 1, то с первого, а если 2, то со второго.

Включив режим показа графика начинаем яростно крутить резисторы из стороны в стороны, наблюдая как наша кривая описывает зигзаги.



Сложно? Нет! Буквально пара команд, а сколько удовольствия!

Скачать проект для AVR Studio [5]

В **АЦП** заложено столько возможностей, поэтому этой статьей рассказ об **АЦП** не оканчивается. В следующий раз я расскажу о простейших алгоритмах фильтрации данных, а также о том как сделать **АЦП** из аналогового компаратора в тех МК, где АЦП не было и в помине.

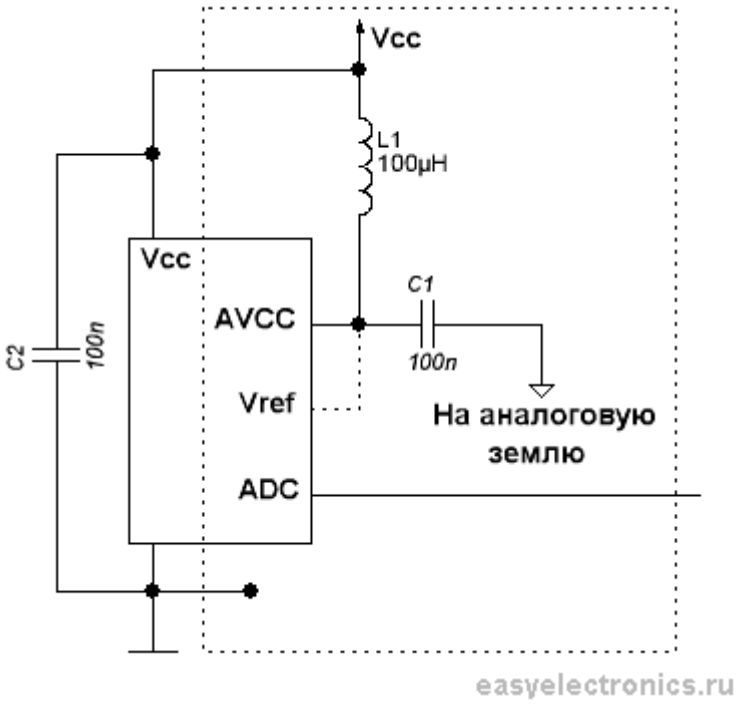
Данная статья была полностью переписана с учетом заточки курса на демоплату Pinboard [2]. Но предыдущий вариант остался в виде архива [6].

Работа с АЦП. Аппаратные средства повышения точности

Фильтрация напряжения
В первую очередь, надо позаботиться о качестве **опорного напряжения**. Ведь выходное значение находится в прямой зависимости от опорного напряжения.

$$\text{АЦП} = (V_{in} * 2^n)/V_{ref}$$

Где, n — разрядность АЦП.
Поэтому желательно использовать специальную микросхему — **Источник Опорного Напряжения**, например, **ADR420** или **REF195**. Стоить они могут недешево — сотни рублей, но зачастую оно стоит того. Прецизионная аналоговая электроника в принципе не дешевая. По началу я тоже пугался ценам в 500-600 рублей за какой то там усилитель. А сейчас ничего, привык :) Впрочем, в фанатизм впадать не стоит. На худой конец, если используется **AVR**, точность которой 2МЗР (младший значащий разряд, если забыл) на десяти битах, то можно не заморачиваться с дорогущими ИОН и городить что попроще, например на **LM336Z-5.0**, включаемых подобно стабилитрону, только куда более точному.



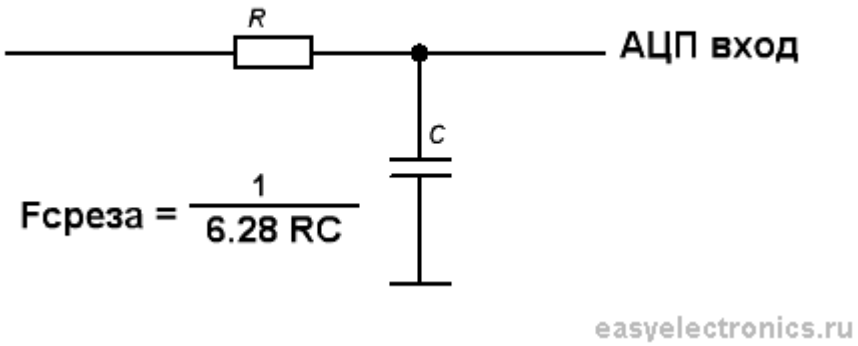
Если совсем лень искать спец деталь, а точность нужна в пределах показометра, то достаточно просто завести на V_{ref} напряжение питания через дроссель в **10..100uH** и повесить кондер емкостью 0.1uF на землю, как можно ближе к выводам. На картинке я нарисовал подачу V_{ref} пунктиром, т.к. в данном случае я привожу пример для AVR, а там AREF можно подключить изнутри кристалла, программно.

Разумеется нужно отфильтровать и питание контроллера, поставив блокирующие емкости из керамики в 0.1uF между Vcc и GND, как можно ближе к выводам МК, чтобы помехи, возникающие при переключении логических уровней внутри МК, подавлялись.

Еще надо сделать **аналоговую землю**, которая будет проходить/окружать всю аналоговую часть, также желательно, чтобы аналоговая земля проходила под микроконтроллером в районе её аналогового порта. Аналоговая земля должна быть как можно более сплошной **и не содержать в себе замкнутых контуров**. С землей цифровой ее следует **соединять в одной точке** и как можно дальше от точных цепей.

Фильтрация сигнала
Разобравшись с питаловом, можно приступать к сигнальной части. Если нам точно известна частота входного сигнала, то мы можем со спокойной душой сделать фильтр и завалить все, что не входит в наш диапазон. Фильтры бывают разные, активные и пассивные, пока я подробно на них останавливаться не буду ~~потому что сам в них толком ни в зуб ногой~~, приведу лишь простенький пример :)

Например, нам надо изменять довольно медленно меняющийся сигнал — напряжение питания. А на провод лезет всякая гнусь: шумы от электропроводки, радио сигналы, помехи от коллекторных двигателей, да мало ли что там в воздухе висит. Чтобы все это похерить, оставив только наш медленно меняющийся сигнал, нужно применить фильтр низких частот. Который бы задавил все быстрые колебания сигнала, оставив только постоянную составляющую.
В простейшем случае, это будет **пассивный RC фильтр**.



Частота среза это такая частота, с которой начинается подавление сигнала. Можно еще и дроссель поставить, последовательно входу.

О фильтрах я еще напишу, а тем кто загорелся фильтровать все подряд я припас **две серьезные книжки по фильтрам**:

- Книжка раз, ИМХО, посложней [1]
- книжка два, вроде попроще [2]

Работа с АЦП. Программные средства повышения точности

Вообще, сграбив сигнал в цифровую форму мы мы можем извращаться с ним как угодно. Методик цифровых фильтраций существует масса и все они основаны на сборе избыточной информации с последующим выделением сигнала. Я приведу для примера лишь один простейший способ — усреднение.

Суть усреднения в том, что у нас есть **статичный** (считаем, что за время измерения сигнал не меняется) сигнал к которому подмешан шум. Шум возникает изза работы транзисторов, из-за колебания опорного напряжения, помех, наведенных на сигнальные линии. Да от чего только он не возникает. Особенность шума в том что он, как правило, **хаотичен**. Так что во время нашего замера может меняться как в меньшую так и в большую сторону. Тут то мы его и прижучим.

Берем и снимаем не одно измерение, а сразу кучу. А потом берем по ним среднее арифметическое. Так как полезный **сигнал у нас константен**, то его составляющая такой и останется, как ее не усредняй, а вот шум изрядно приглушит. И чем больше выборок мы сделаем, тем сильней задавит шумовую составляющую. Западло этого метода очевидно — резко снижается скорость обработки. Так как вместо одной выборки нам приходится делать серию и объявлять ее как одну, но это неизбежное зло.

В качестве демонстрации метода я приведу пример усреднения. Программка простая, хватает 64 выборки, усредняет их и отправляет по UART. Сразу отмечу тот факт, что для эффективного подавления шума нужно чтобы частота выборок была ниже частоты всяких паразитных колебаний (вроде 50Гц наводок от сети) раза в два три, иначе у нас эти колебания благополучно пролезут как полезный сигнал. А еще число выборок нужно брать кратным двойке, чтобы можно было делить простым сдвигом. Впрочем, смотрите на код, там будет более понятно. Весь код я выкладывать не буду, только главный файл. Все инициализации АЦП и UART, а также ряд служебных процедурок я оставлю за кадром. Если интересно, то вы всегда можете **скачать проект** [1] и посмотреть сами. Сбор числа у меня идет в прерывании от АЦП, а деление в прерывании по передаче. Так минимизируется число действий выполняемых процом. Хотя растягивание прерываний это не есть гуд. Но городить флаговую операционную систему мне тут впадлу, впрочем, дойдет и до нее время. А пока смотрите в код:

```
1                   .include "m16def.inc"  ; Используем ATMega16
2                   .include "define.asm"  ; Наши все определения переменных тут
3                   .include "macro.asm"   ; Все макросы у нас тут
4                   .include "vectors.asm" ; Все вектора прерываний спрятаны в этом файле
```

```
5   ;=============================================================================
6
7                   .ORG    INT_VECTORS_SIZE
8   Reset:          OUTI    SPL,low(RAMEND)
9                   OUTI    SPH,High(RAMEND)
10
11  ;=============================================================================
12                  .include "init.asm"           ; Все инициализации тут.
13  ;=============================================================================
14  Main:           SEI                           ; Разрешаем прерывания.
15
16
17  ;Главный цикл пуст, весь экшн в прерываниях.
18
19
20                  RJMP    Main
21
22  ;=============================================================================
23  ; Interrupts
24  ;=============================================================================
25  ; Прерывание от UART на передачу
26  UART_TX:        PUSH    R16             ; Нычим регистры в стеке
27                  IN      R16,SREG
28                  PUSH    R16
29
30
31  ; Сумму мы нажрали в прерывании по АЦП, а делить ее для усреднения будем уже тут
32                  CLC             ; Сбрасываем флаг переноса
33                  ROR     SUMH    ; И сдвигаем значение вправо сначала старший
34                  ROR     SUML    ; потом младший. Через флаг переноса.
35
36                  CLC             ; Всего нам надо 6 сдвигов 2^6=64
37                  ROR     SUMH    ; Что будет эквивалентно делению на 64
38                  ROR     SUML
39
40                  CLC
41                  ROR     SUMH
42                  ROR     SUML
43
44                  CLC
45                  ROR     SUMH
46                  ROR     SUML
47
48                  CLC
49                  ROR     SUMH
50                  ROR     SUML
51
52                  CLC
53                  ROR     SUMH    ; Теперь уменя готовое 10 байтное усредненное число
54                  ROR     SUML    ; Но 10 байт мне нафиг не уперлись. Куда мне их совать?
55                                  ; Поэтому я отбрасываю еще два младших бита, превращая
56  число
57  ; в восьмибитное. Делаю я эту нехитрую операцию тем же самым сдвигом через перенос
58
59                  CLC
60                  ROR     SUMH
61                  ROR     SUML
62
63                  CLC
64                  ROR     SUMH
65                  ROR     SUML    ; Все, теперь можно забыть про старший байт, он равен
66  нулю
67                                  ; А весь жыр теперь находится в младшем байте SUML
```

```
68
69
70
71                OUT     UDR,SUML ; Его то мы и отрыгиваем в UART. О переполнении буффера
72  можно
73                                 ; Не париться, так как отправка идет исключительно по
74   ; прерыванию, а значит пока байт не ушел никаких левых засылок не будет.
75
76                POP     R16             ; Достаем из стека сныканое барахло.
77                OUT     SREG,R16
78                POP     R16
79                RETI                    ; Выход из прерывания
80
81
82  ;=============================================================
83  ; Прерывание от UART на прием
84  UART_RX:      PUSH    R16             ; Прячем в стек регистры
85                IN      R16,SREG
86                PUSH    R16
87
88                IN      R16,UDR         ; Берем пойманый байт
89                CPI     R16,'s'         ; И анализируем. Если S - значит стоп
90                BREQ    StopADC
91
92                CPI     R16,'r'         ; Если r значит - запуск.
93                BREQ    RunADC
94
95  ; Запускаем АЦП, на непрерывное преобразование и включаем UART на передачу
96  RunADC:       OUTI    UCSRB,(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)
97                OUTI    ADCSRA,(1<<ADEN)|(1<<ADIE)|(1<<ADSC)|(1<<ADFR)|(3<<ADPS0)
98                OUTI    UDR,'R'
99                RJMP    ExitRX
100
101 ; А тут останавливаем и АЦП и выключаем UART иначе он болезный будет гадить
102 ; по прерыванию до тех пор пока питалово не кончится.
103 StopADC:      OUTI    ADCSRA,(0<<ADEN)|(0<<ADIE)|(0<<ADSC)|(0<<ADFR)|(3<<ADPS0)
104               OUTI    UCSRB,(1<<RXEN)|(0<<TXEN)|(1<<RXCIE)|(0<<TXCIE)
105
106 ExitRX:       POP     R16                     ; Достаем из стека данные наши
107               OUT     SREG,R16
108               POP     R16
109               RETI                            ; Выходим в главный цикл
110
111
112 ;=============================================================================
113 ; Прерывание от АЦП - обработка завершена
114 ADC_OK:       PUSH    R16                     ; Сохранить регистры в стеке
115               PUSH    R17
116               IN      R16,SREG        ; Сохранить SREG
117               PUSH    R16
118
119 ; Определяем ряд макросов для удобства использования
120               .def    ADSH    = R15   ; Старший байт суммы
121               .def    ADSL    = R14   ; Младший байт суммы
122               .def    ACT     = R13   ; Счетчик усредняемых значений
123               .def    SUMH    = R12   ; Выходной регистр старший
124               .def    SUML    = R11   ; Выходной регистр младший
125
126
127               IN      R16,ADCL        ; Взять значение из АЦП
128               IN      R17,ADCH
129
130 ; Теперь пару R17:R16 надо просуммировать с их n-1 значением. При первой итерации
```

```
131 ; тут ноль. При второй уже будет сумма с первой выборкой и так далее.
132
133
134                 ADD     ADSL,R16        ; Младший байт суммируем без переноса
135                 ADC     ADSH,R17        ; Старший байт суммируем с учетом переноса
136
137 ; В результате, в ADSH:ASSL будет накопленна сумма ADSH:ASSL(n) + ADSH:ASSL(n+1)
138 ; Максимальное значение для 10битного АЦП - 1024, в то время как в 16разрядах
139 ; Помещается 65536 значений, 65536/1024 = 64. Поэтому переменная ACT должна быть
140 ; Заранее инициализирована числом 64
141
142                 DEC     ACT     ; Считаем сколько уже выборок мы сделали
143                 BRNE    Exit            ; Если это не последнее слагаемое, то выход
144
145                 MOV     SUML,ADSL       ; Сливаем выходное значение в регистр
146                 MOV     SUMH,ADSH       ; младший и старший байты
147
148                 CLR     ADSL    ; Потом мы очищаем регистры суммирования
149                 CLR     ADSH    ; Чтобы новый результат был с нуля
150
151                 LDI     R16,64  ; Заполняем счетчик новым числом
152                 MOV     ACT,R16
153
154 Exit:          POP     R16             ; Достаем из стека все как было.
155                 OUT     SREG,R16
156                 POP     R17
157                 POP     R16


                    RETI                    ; Возвращаемся.
```

AVR. Учебный Курс. Отладка программ. Часть 1

У каждого случалась такая ситуация — программа вроде бы написана, даже компилится, но не работает. Почему не работает? Дак все же просто — в ней есть лажа!

Процесс избавления программ от лажи называется, соответственно, отлаживанием. И часто этот процесс длится куда дольше разработки, особенно в хитрых случаях, с привлечением внешней периферии.

В очередном цикле статей я постараюсь описать как можно более подробно методы, применяемые при отладке.

Метод 0. Тупление в код (Аналитический)

К моему великому удивлению, данный метод является наиболее популярным в народе и, а у начинающих порой единственным.
Видимо сказывается засилье всяких высокоуровневых языков вроде ПОХАПЭ или Си, где такое вполне может и проканать. Там да, можно пофтыкать несколько минут в исходник, глядишь да найдешь где накосячил.

И по наивности, не иначе, новички пытаются этот метод применить к своим ассемблерным программам.

И вот тут мозг срывает напрочь. С непривычки голова пухнет от попытки удержать в памяти состояние регистров, флагов, попыток просчитать куда пойдет ядро на следующем шаге.

Из этого же лезет народ убежденность в том, что ассемблерные программы сложны в написании и отладке.

Хотя я, в свое время, изучал ассемблер вообще без компа — не было его у меня. Тетрадка в клеточку, команды i8008 в столбик. А потом и Z80 с его божественным ассемблером. И опять без отладчиков, аналитически. Ляпота! Но вот когда я сел за ассемблер 80C51, в первую же очередь я нашел нормальную IDE с эмуляцией — Keil uVision. А эра x86 прошла под знаменем Borland Turbo Debugger и TASM. Когда моя первая 128 байтная интруха полыхнула по экрану пиксельным пламенем клеточного автомата… да ощущения были еще те. Где то ее сорцы валяются, надо поискать.

В написании может быть, но вот в отладке нифига подобного. Так как нечего тут думать. Ассемблер это как лопата — берешь и копаешь, а не думаешь как там поршни и трансмиссия в экскаваторе крутится.

А вот когда уже есть некоторый опыт ковыряния с ассемблером, когда всякие ветвления-переходы-адресации разруливаешь в уме, вот тогда аналитическим тупняком можно сходу искать баги. Но это для мастеров низкоуровневого кунг-фу. Поначалу, быстрей и проще пройтись трейсером по коду.

Под каждый стиль написания кода свои инструменты отладки. Поэтому переходим к другому методу.

Метод 1. Лопата — бери и копай (Трассировка)

Не можешь понять как будет вести код? Как сработают условия? Как пойдет ветвление? Что будет в регистрах?

Что тут думать? Выполни в эмуляторе и все! Какая-нибудь AVR Studio идеально для этого сгодится.

Компилируй программу и запускай на выполнение, а дальше выполняй ее пошагово, наблюдая за регистрами, флагами, переменными. Да за чем угодно, все в твоей власти.

Но тут есть ряд тонкостей, для кого-то очевидных, а кому-то и в голову не придет. Поэтому сыграю в Капитана и разжую их все.

Трассировка

Активно используй не только пошаговую трассировку (F11), но и такие удобные фичи как трассировка до курсора (Ctrl+F10), позволяющая сразу же выполнить весь код до курсора. Правда может застрять в каком нибудь цикле по пути. Но можно нажать на паузу и вытащить трассировщик из цикла вручную (выставив флаги так, чтобы произошел нужный нам переход).

Обход ненужных условий

Условия, встречающиеся на пути, довольно легко обходятся путем установки вручную нужных флагов. А если логика отлаживаемого участка от этого не пострадает, то можно и временно закомментить мешающий код. Если до нужного участка кода предстоит продраться через несколько десятков условий, то можно облегчить себе задачу, воткнув сразу же после всех нужных инициализаций секцию DEBUG

```
1 ;-----DEBUG
2         JMP KUDA_NADO
3 ;----------
```

И все. И не напрягаясь отладить нужный участок, а потом удалить этот переход, чтобы вернуть логику программы в прежнее русло.

Генерация прерываний

Если нужно отлаживать прерывание, то необязательно этого прерывания дожидаться. Помни, что вызов прерывания делается установкой флага в соответствующем регистре. Т.е. хочешь ты сэмулировать прием байта по UART — тыкай на бит RXC регистра UCSRA и вот у тебя на следующем же шаге произошло прерывание. Разумеется если оно разрешено локально и глобально.

Если хочешь запустить прерывание в случайный момент. То нет ничего проще — запусти программу на исполнение (F5) через пару секунд тупления в монитор нажми паузу (Ctrl+F5), тычком по нужному биту сгенерь прерывание, а дальше пошагово влезь в обработчик и позырь что там происходит. После чего проверь, что выход из прерывания произошел корректно и пусти прогу пастись дальше.

Ускоряй процессы

Многие процессы на трассировке идут весьма длительно. Например, в режиме скоростного прогона (F5) эмуляция секунды на моем компе (AThlonXP 1300+/512DDRPC3200) идет около минуты, если не больше. Разумеется ждать столько западло.

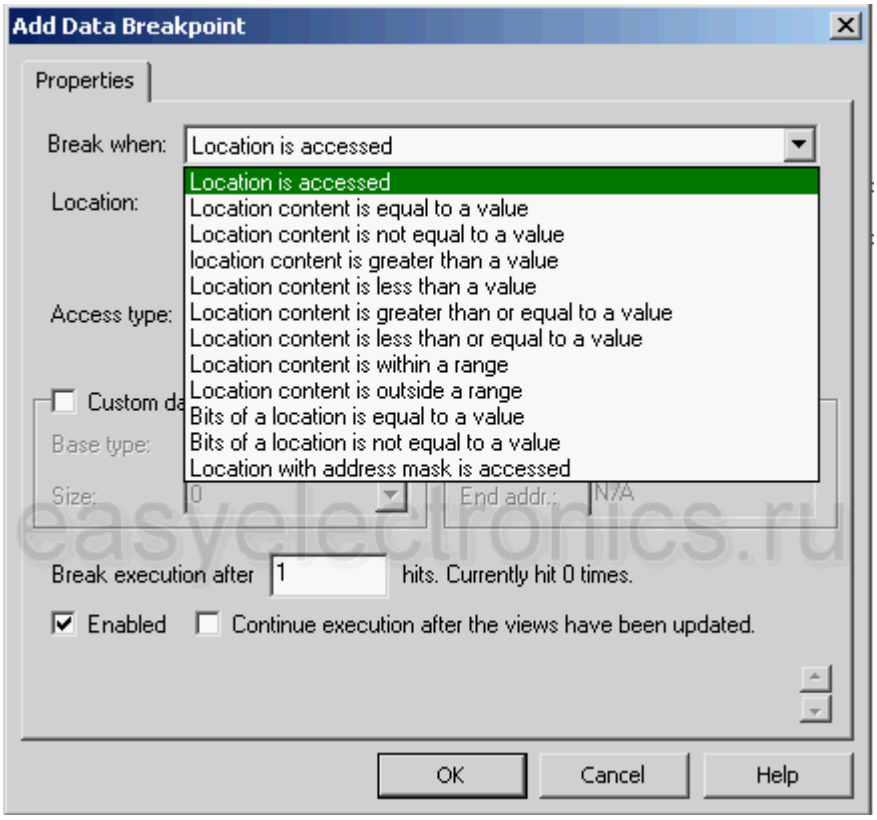
Но зачем ждать? если алгоритм задержки верен, то что нам мешает взять и уменьшить время с секунд до десятков миллисекунд?
Если задержка сделана на таймере, то можно временно уменьшить предделители таймеров в 1. Или даже выставить другие значения уставок. Главное, не забыть вернуть все как было.

Остановка по требованию
Это в маршрутках остановок «Здеся» и «тута» не существует. В нашей же симуляции мы можем выгрузиться в пошаговый режим где угодно, хоть посреди МКАД в час пик. Достаточно применить волшебный щелчок по почкам.

Используй брейкпоинты или точки останова. Их можно ставить на произвольные участки кода и в режиме запуска на автомате (F5) симулятор сам воткнется в ближайший брейкпоинт. А дальше вручную, пошагово, проходим трудный участок. Выясняя почему он работает не так.

Причем брейкпоинты можно ставить не только на участки кода, но и на события происходящие в регистрах. Это, так называемые, Data Breakpoint. Ставятся они из Debug-NewBreakPoint-DataBreakPoint

А там на выбор событий тьма:



Например, хочешь ты поймать событие когда кто то стукнулся в ОЗУ и нагадил, запоров значение. Кто такой поганец? Стек? Где то в коде опечатался и не заметил? Индексный регистр не туда залез?

Чем гадать и тупить в код лучше поставить бряк. Например, загадилась переменная CCNT объявленная в RAM:

```
1 ; RAM =============================================================
2         .DSEG
3 CCNT:   .byte   4
4 TCNT:   .byte   4
5 ; FLASH ===========================================================
```

Выбираешь тип события, Location is Acessed (доступ к обьекту) в поле location выбираешь объект за которым будем следить В выпавшем списке будут ВСЕ твои символические имена используемые в программе. В том числе и наши CCNT и TCNT. Надо только внимательно поискать — они там не в алфавитном порядке, а черт знает в каком.

Дальше выбираешь тип доступа к переменной (Access Type) — чтение или запись, а может и то и другое. Также можно указать тип данной переменной или ее байтовый размер.

А еще можно в памяти разметить колышками Start Addr — End Addr (в графе Custom Scope) делянку, где наша конопля растет. И как только туда кто сунется — алярм, ловить и пинать по почкам.

А через контекстное меню, можно по быстрому, на любой регистр, ячейку ОЗУ/ПЗУ/EEPROM повесить Data breakpoint. При этом он сразу же появляется в окошке Breakpoints&TracePoints и там уже можно подправить ему конфигурацию как нужно.

Осталось только заполнить графу хитов Break Execution after… Т.е. после скольких событий останавливать трассировку. По умолчанию стоит 1. Но если, например, нам надо пропустить несколько сотен итераций цикла, а на сто первом посмотреть что происходит, то указываем число хитов в 100 и жмем запуск, не страдая фигней на промотке этих итераций.

Есть там еще одна интересная галочка — Continue executions after views have been updated. Она превращает бряк в информер. Думаю ты уже замечал, что когда студия гонит код в режиме выполнения (F5), то данные в окошках периферии и регистрах в реальном времени не меняются. Чтобы увидеть изменения надо поставить программу на паузу.
Так вот, брейкпоинт-информер нужен для принудительного обновления этих значений. Т.е. программа на нем не останавливается, а только лишь обвновляет данные в окошках. Что позволяет динамически наблюдать как данные ползают по памяти, как щелкают таймеры, как тикают регистры.

Своего рода автоматическое пошаговое выполнение (Alt+F5), но обновление не по каждой команде, а по условию. Да, превратить обычный путевой бряк в информер можно, но для этого надо открыть окно управления брейкпоинтами View-Toolbars-Breakpoint&TracePoint. И там уже, найдя наш путевой брейк, даблкликом по нему залезть в свойства.

Там же бряки можно оперативно включать-выключать, что очень удобно. Выключать и включать их также можно по F9.

Пускай на самотек
Если программа в железе ведет себя как то не так, то может у ней крышу сорвало? Когда пишешь на ассемблере (да и на Си тоже, но там поймать сложней) легко ошибиться со стеком или индексным переходом. А сразу не поймаешь. Далеко не факт, что срыв найдется при первом или втором пошаговом прогоне. В этом случае я просто жму запуск симуляции и ухожу нарезать колбаски и чаю налить. Прихожу — жму паузу. Если прога идет по своему циклу, значит все в порядке. Если же произошел срыв, то это будет сразу же видно — трассировщик сойдет с ума и выбросит тебя из исходного кода в дизассемблер, где будет видно, что происходит что то явно не из нашей оперы. Например, выполнение кода там где его нет — за пределами программы. Если такое случилось надо внимательно проверить возможные пути срыва. Наставив брейков на входы-выходы из прерываний, перед и после индексных переходов и так далее.

Еще тут может помочь режим автошага (Alt+F5) нажал его и студия сама начала шустро тикать тактами, сразу же показывая что происходит. Тебе же остается сидеть и тупить в этот телевизор, глядишь найдешь глюк. Пару раз я таким способом отлавливал ошибки атомарного доступа, вылезавшие только в полнолуние и исключительно по спец мантре.

Глядим внимательно
Несмотря на то, что можно тупить в регистры, порты или напрямую в память, делать это все равно неудобно — глаз замыливается разглядывать в этой каше наши значения. Даже несмотря на то, что при изменении они подкрашиваются красным. Тут нам поможет очередной инструмент студии Watch (вызывается по Alt+1). Ватч это гляделка. С ее помощью можно наши выделенные переменные пихать в отдельные окошки, где за ними удобно наблюдать.

Навести гляделку можно на что угодно — на регистр, на ячейку памяти, на порт. При этом мы можем выбирать тип отображения (DEC,HEX,ASCII), что бывает очень удобно.

А если мы отлаживаем Сишный код, то watch умеет показывать структуры и массивы не в виде кучи барахла в памяти (какими они на самом деле и являются), а красиво раскладывая все по полочкам.

Эмуляция периферии
Внутреннюю периферию отлаживать проще простого — все прерывания вот они, свисают в IO регистрах, дергай не хочу. Регистры там же. А вот внешняя…

Тут все плохо. Дело все в том, что Студия не имеет понятия о том, что есть за ее пределами. Поэтому либо отлаживать периферию ручным тыком по битам в портах (что ужасно муторно), либо юзать примочки вроде HAPSIM или StiGen от ARV [1]. Да, пользуясь моментом, рекомендую прошуршать по сайту arv.radioliga.com [2] — много интересных штуковин. Проблем они всех не решают, но лучше чем ничего.

Трассировка по Сишному коду
А тут все весело. Дело в том, что оптимизатор может так залихватски перелопатить код, что на исходный код он будет походить весьма и весьма отдаленно. Т.е. ты грузишь переменную в одном месте, а компилятор решил это сделать в самом начале программы. Работать то будет так как ты сказал, но вот отлаживать это… Некоторые участки кода вообще будут перепрыгиваться.

Т.к. оптимизатор все заоптимизировал за счет других строк, а промежуточные варианты тебе в виде отчета забыл предоставить.

В общем, я понимаю почему трассировка среди тех кто пытается писать на Си не прижилась. В том виде в каком ее видишь в первый раз пользоваться ей не хочется совершенно. В самом деле, кому охота связываться с дебильным симулятором? Проще уж в код тупить, там хоть какая то логика есть.
Но не стоить хоронить трассировку по высоким языкам раньше времени. Если отбросить приколы с выполнением некоторых участков кода, то все еще вполне ничего. А если выключить на время оптимизацию (параметр -O0), то вообще самое то. Правда отлаживать совсем без оптимизации я бы не советовал.

Т.к. с оптимизатором там есть свои приколы и грабли. И при несоблюдении ряда правил (volatile, пустые циклы и прочие фишечки), код который прекрасно работает без оптимизации на -Os с треском разваливается на куски.

Но общую логику работы программы отследить можно. А учитывая умные гляделки, бряки, информеры… так вообще сказка!

Но это далеко не все методы отладки. А так, самая малость. Однако трассировкой можно выловить 90% проблем связанных с внутренней периферией и алгоритмом работы кода на уровне ядра.

Впереди же будут описаны еще реалтаймовые способы на реальном железе — дебаг выводы, развлекухи с осциллографом, облизывание на логический анализатор и JTAG мониторы.
Не обойду вниманием и симуляторы вроде PROTEUSa, хотя я с недавних пор предпочитаю ими не пользоваться — мне своих глюков хватает.

AVR. Учебный Курс. Отладка программ. Часть 2

Метод 2. Моргалки (Работа портами Ввода-вывода)

Трассировка и аналитика это все замечательно, но когда мы начинаем отлаживать что то внешнее, то тут трассировка нам поможет мало. Т.к. глючить может не внутри, а снаружи.

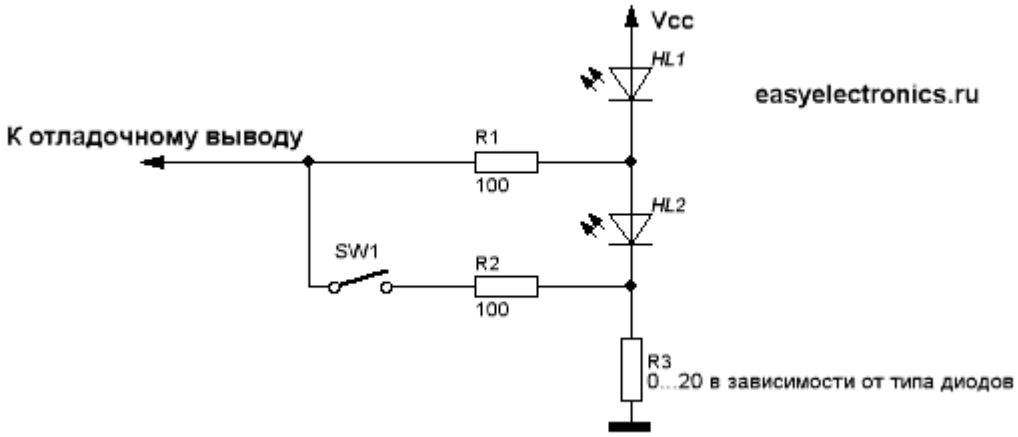
Либо у нас будет не глюк, а банальное непонимание ситуации. Фразу из даташита, например, неправильно перевел или глаз замылился. Что то забыл или не туда припаял. Да мало ли как можно накосячить.

В общем, правильный алгоритм может работать из-за кривой аппаратной реализации. А чтобы понять где косяк надо проследить и знать как это работает внутри. В общем, заглянуть в код программы.

Причем нам не обязательно точно следить за тем, что там происходит покомандно — это мы или трейсом или туплением отладим. Интересней знать куда ушла прога.

Самым простым методом тут будет отладочный вывод. Все просто — мы берем и определяем один из выводов под отладку. Причем даже если у нас какая-нибудь крошечная тинька и все выводы расписаны буквально по нескольку раз, то все равно можно найти один лишний вывод — достаточно закомментить код который его использует, но в данный момент его правильность нас не интересует.

А у вывода может быть три состояния — Hi, Lo и Hi-Z. Так что рекомендую скреативить такой вот пробничек:



Тогда если у нас Hi-Z то будут тускленько гореть оба диода. Ну, а на соответствующий уровень свой диод в гордом одиночестве.

А в код пихаем дебажные секции вида:

```
1 ;Set Hi
2         SBI     DEBUGDDR,DEBUG
3         SBI     DEBUGPORT,DEBUG
1 ;Set Lo
2         SBI     DEBUGDDR,DEBUG
3         CBI     DEBUGPORT,DEBUG
1 ;Set Hi-Z
2         CBI     DEBUGDDR,DEBUG
3         CBI     DEBUGPORT,DEBUG
```

Можно их в макросы завернуть. Удобней будет.

По состоянию дебаг вывода мы можем увидеть в каком месте программа прошла. А комбинируя нашу троичную систему получим целых 9 возможных комбинаций на 2 вывода.

Однако если у нас надо отследить последовательность, то мы можем и не успеть разглядеть переимгивание светиков. Но никто не мешает воткнуть в код после смены состояния дебагвывода тупую задержку вида:

```
1          PUSHF                   ; Макрос сохранения флагов
2          PUSH    R16
3          PUSH    R17
4          PUSH    R18
5
6          LDI     R16,255 ; Грузим три байта
7          LDI     R17,255 ; Нашей выдержки
8          LDI     R18,255
9
10 Dloop:  SUBI    R16,1           ; Вычитаем 1
11         SBCI    R17,0           ; Вычитаем только C
12         SBCI    R18,0           ; Вычитаем только C
13
14         BRCC    DLoop           ; Если нет переноса - переход
15
16         POP     R18
17         POP     R17
18         POP     R16
19         POPF                    ; Макрос восстановления флагов
```

Оформляется как прерывание для того, чтобы иметь минимальное влияние на код. Можно тоже оформить в виде макроса и втыкать его одной строчкой. Запрет прерываний опционально.

Либо вставлять затуп:

```
1         RJMP    PC
```

Контроллер на этой команде встанет и дальше никуда не пойдет (правда если включен вачдог, то он ребутнется по вачдогу).

А можно же сразу после задержки перевести отладочный вывод в режим входа PullUp и просить кнопку пробника. Если нажата, можно идти дальше. Можно и без задержки, сделать, но надо смотреть, чтобы дребезг не помешал.

```
1 ;STOP DB_BTN
2         CBI     DEBUGDDR,DEBUG
3         SBI     DEBUGPORT,DEBUG
4
5         SBIC    DEBUGPIN,DEBUG
6         RJMP    PC-1
```

Можно просто наставить таких кнопочных чекпоинтов. И пускать программу по ним своей могучей волевой рукой. А пока программа тупит на задержках или ожиданиях кнопок, можно спокойно замерить напряжения/пощупать лог уровни, состояние других выводов, да просто подумать.

Разумеется, если в схеме уже присутствуют светодиоды и кнопки, то и пробник тут не нужен — обойдемся имеющимися средствами — переназначив их на время отладки конкретного участка.

Правда в случаях отладки асинхронных устройств, работающих на четких временных выдержках данная затея может не прокатить, но для них есть другие варианты, о них позже.

AVR. Учебный Курс. Отладка программ. Часть 3

Метод 3. USART (Работа с последовательными интерфейсами)

Пожалуй самым популярным отладочным интерфейсом является все же USART. Во-первых, он поддерживается аппаратно почти всеми микроконтроллерами. Во-вторых, он прост в использовании и требует всего один/два сигнальных провода, а в третьих, для связи с компом не надо городить никаких специфичных девайсов. В худшем случае UART-USB или UART-RS232 конвертер на FT232RL или MAX232.

Пользоваться им проще простого — в любой момент, когда нам это захочется, мы берем и отправляем нужный байт по этому интерфейсу. При этом достаточно заранее его настроить — стандартная инициализация UART:

```
1  ; Usart INIT
2                  .equ    XTAL = 8000000
3                  .equ    baudrate = 9600
4                  .equ    bauddivider = XTAL/(16*baudrate)-1
5
6  uart_init:      LDI     R16, low(bauddivider)
7                  OUT     UBRRL,R16
8                  LDI     R16, high(bauddivider)
9                  OUT     UBRRH,R16
10
11                 LDI     R16,0
12                 OUT     UCSRA, R16
```

```
1 ; Прерывания запрещены, прием-передача разрешен.
2         LDI     R16, (1<<RXEN)|(1<<TXEN)|(0<<RXCIE)|(0<<TXCIE)|(0<<UDRIE)
3         OUT     UCSRB, R16
```

Маркеры
Всегда полезно знать где наша программа шляется в данный момент. Конечно, можно зажигать светодиодики, как в прошлом методе. Но когда точек много, то выводов не напасешься на эту затею. Или затрахаешься перекомпилировать. Проще всего натыкать в код маркеров, наступая на которые наш МК будет отрыгивать в терминал свое местоположение. Терминальный клиент будет писать лог и ничего не проморгаешь.

Для отправки надо записать наше маркерное число в регистр UDR, пусть это будет код буквы «A».

```
1         LDI     R16,'A'
2         OUT     UDR,R16
```

Я даже проверку на занятость UDR не делаю. Для одиночного байта, если у нас отправки по UART нет и участок не зациклен, этого вполне достаточно — регистр будет свободен. Разумеется в реальной программе такого делать не следует, но для отладочной затычки вполне сойдет.

Можно нафаршировать весь код такими вот затычками, но с разными буквами и смотреть в терминалке в каком порядке программа выполняется по выдаваемому слову. Правда учитывайте тот момент, что если данные будут сыпаться в терминалку быстрей чем она ее может прожевать, то регистр UDR захлебнется и у нас будет полная лажа. В этом случае можно сделать затычку чуть сложней, с ожиданием освобождения UDR

```
1         LDI     R16,'A'
2         SBIS    UCSRA,UDRE              ; Пропуск если нет флага готовности
3         RJMP    PC-1                    ; ждем готовности - флага UDRE
4         OUT     UDR, R16                ; шлем байт
```

Можно в макрос это запихать:

```
1         .MACRO  DEB_SEND
2         PUSH    R16
3         LDI     R16,@0
4         SBIS    UCSRA,UDRE
5         RJMP    PC-1
6         OUT     UDR,R16
7         POP     R16
8         .ENDM
```

Заодно R16 в стеке спрячем, так что макрос можно юзать где угодно, словно команду:

```
1         DEB_SEND 'A'
```

Но это затормозит выполнение программы, могут полететь тайминги. Так что если у вас есть какие то участки жестко зависимые от времени (скажем тайминги протокола 1-wire), то данная затычка даст сбой.

Выводить можно любую инфу. Например содержание нужных регистров, содержание переменных, ячеек памяти, состояния битов регистров периферии. Что хотим поглядеть — то и тащим. Надо только загрузить это в UDR. А поскольку там могут быть произвольные значения, то глядеть их лучше через терминалку способную показывать в хексах. Например, моя любимая Terminal v1.9b.

Также полезно выбрасывать в терминал маркеры прохождения кода. Чтобы оценить правильность переходов и логики работы.

Еще настоятельно рекомендую заиметь спец литеру отправляемую в терминал сразу же после инициализации USART, еще до перехода к основной программе. У меня это обычно буква «R». От «Reset». Это позволяет поймать момент перезагрузки контроллера.

Например, глючила у меня программа. Все работало нормально, но при попытке принять байт все начинало странно глючить. Все облазил, код весь до дыр проглядел. По коду все должно работать… Наконец воткнул в код OUTI UDR,’R’ еще до вхождения в main. Опаньки — а контроллер то сбрасывается!

А почему контроллер может сбрасываться? Сбой по питанию я исключил сразу — не было там ничего такого, что бы могло дергаться. RESET был подтянут через 10кОм и особо тоже не вихлялся. Что еще?

Срыв стека, если произошел переход за конец кода, тоже похож на сброс — программа уходит за конец памяти и возвращается к нулевому адресу. Прошарил в отладчике и этот момент — нет срыва. Но может срыв не сразу, а спустя несколько итераций? Поставил .ORG на предпоследний адрес флеша и прописал туда маркер — не вылазит, значит срыва стека нет.
Да и код был прост как мычание, нечему там глючить. Накидал отладочных маркеров по коду, засылающих байты в строго определенном порядке — в терминалке высветилась фраза «УБЕЙ СИБЯ АПСТОЛ» — значит прога работает четко по алгоритму.

Заглянул под панельку… Вот в чем засада — медная ворсинка от провода МГТФ попала между RXD и RST (На меге8 они совсем рядом) и при приходе байта в порт дрыганья на RXD сбрасывали контроллер. Соблюдайте чистоту на рабочем месте! Не допетри я тогда поставить на перезагрузку маркер, так я бы до утра ковырялся с кодом. А так я резко сузил круг возможных косяков и ушло не более десяти минут на отлов этого бага.

Управление кодом, добыча произвольных данных
Но не едиными маркерами сыт боец USART'овой отладки.

Мы же можем не только слать, но и принимать! А принятое обрабатывать! Так что нам мешает делать полноценные брейкпоинты?

Скажем, такие:

```
1         PUSH    R16
2         SBIS    UCSRA,RXC
3         RJMP    PC-1
4         IN      R16,UDR         ; Читаем UDR, чтобы сбросить RXC флаг
5         POP     R16
```

Данный код является банальным бряком, т.е. программа на ней встанет как вкопанная до тех пор, пока мы не зашлем ей байт в терминалку. Удобно, особенно когда осциллографа нет под рукой, можно, например, спокойно мультиметром уровни замерить на выводах, а потом пустить прогу до следующего чекпоинта.

В сочетании с маркерами дает еще и информацию о том, где мы встали. Но можно же и большее заиметь!

Что нам мешает сделать небольшой обработчик команд? Скажем, если мы послали R — выдаст значение нужного регистра, если M — ячейки памяти, I — байта из IO, а G — пойдет дальше:

Пример макроса может быть таким:

```
1          .MACRO  DEB_CMD
2          PUSH    R16             ; Сохраняем регистр и флаги в стек
3          IN      R16,SREG
4          PUSH    R16
5
6          SBIS    UCSRA,RXC
7          RJMP    PC-1
8          IN      R16,UDR         ; Читаем UDR, чтобы сбросить RXC флаг
9
10                                 ; Определяем что там пришло
11         CPI     R16,'R'
12         BREQ    PC+0x07         ; BREQ REGISTER
13
14         CPI     R16,'M'
15         BREQ    PC+0x07         ; BREQ  MEMORY
16
17         CPI     R16,'I'
18         BREQ    PC+0x09         ; BREQ  IO
19
20         CPI     R16,'G'
```

```
21          BREQ    PC+0x0A         ; BREQ GONEXT
22
23
24          OUT     UDR,@0          ; REGISTER - отправляем в UDR значение регистра
25          RJMP    PC+0x0008       ; Проверка на пустоту UDR тут скорей всего не нужна.
26                                  ; Т.к. пока впечатаешь ручкам в терминалку байт, наверняка
27                                  ; все что хотело уйти уже уйдет и UDR опустошится.
28
29          LDS     R16,@1          ; MEMORY - шлем значение из памяти
30          OUT     UDR,R16
31          RJMP    PC+0x0004
32
33          IN      R16,@2          ; IO - шлем значение из порта ввода-вывода.
34          OUT     UDR,R16
35
36          POP     R16             ; GONEXT - достаем все сохраненное из стека и идем дальше
37          OUT     SREG,R16
38          POP     R16
39          .ENDM
```

Как видишь, макрос массово использует относительные переходы (иначе при вставке двух макросов метки будут дублироваться). Так что модифицировать его нужно очень осторожно. Высчитывая команды. Но есть спрособ проще — написать его вначале с обычными метками, скомпилить, запустить отладку, потом открыть дизассемблер (view->disassembler) и поглядеть там смещения.

Используется просто — вставляешь в код такую штуку:

```
1         DEB_CMD R15,0x0000,PORTD
```

И при попадании в это место МК остановится и будет ждать приказа. По команде G — пойдет дальше. По команде R — даст содержание регистра R15, по команде M — содержание ячейки ОЗУ с адресом 0x0000, а по команде I — значение из порта PORTD. Порты, регистры, ячейки памяти вписываешь те которые хочешь посмотреть. Если нужно что то одно, то пишешь в ненужные поля любые подходящие значения. Ну или делаешь узкоспециализированные макросы под память, регистр и порт ВВ.

Можно еще прерывания запретить, чтобы стопор был полный, но не забывай, что всякие таймеры продолжают тикать самостоятельно вне зависимость от того работает МК или зациклен в ожидании команды. А еще, если включен WATCHDOG, то он может и за задницу укусить. Поэтому в макрос можно и команду WDR добавить, в цикл ожидания байта.

Более того, никто не запрещает написать простейший (или сложный) командный обработчик с шахматами и поэтессами, позволяющий указывать непосредственно в передаваемой команде какой ресурс надо забрать. Или что и в какой регистр записать, куда перейти… В общем, этакий микро SoftICE (если кто еще помнит этот легендарный отладчик).

А если дружишь с какими-нибудь Дельфями, то можешь снаружи и отладчик написать, который через этот супервизор будет выковыривать все нужные данные по USART и раскладывать их красивыми рядами на панельках, дабы было любо-дорого глядеть. Например Николев в своем отладочном модуле так и сделал, правда он через SPI работал, но не суть важно [1].

Оставляю это на самостоятельное изучение :)

На Си все пишется примерно в том же ключе. Можешь функции забабахать под это дело. Можешь макросы, тут по желанию.

AVR. Учебный Курс. Отладка программ. Часть 4

Продолжаем трактат об отладке программ. На этот раз в бой идут одни старики.

Осциллограф
Очень часто хочется в динамике поглядеть как работает программа. Особенно если ее структура сложней чем просто суперцикл. Если там конечные автоматы на прерываниях или разделение задач на флаговом автомате/очередном диспетчере, то аналитическое тупление в код и прогоны в отладчике мало что дадут — наслоение прерываний, процессов в очереди, стадии и взаимодействие разных конечных автоматов взорвут мозг кому угодно. А если еще программа не написана с нуля, а собрана из кучи чужих исходников, да даже если из своих, но древних и уже забытых.

Тут очень хорошо помогает осциллограф. Но сначала надо как то вывести сигнал на него. Для этого подойдет любая ножка которой мы можем дрыгать. Обычно для отладки оставляю парочку. На худой конец, можно на время отладки переназначить какой-либо вывод и использовать его.

Для примера возьму ту программу, что идет в доке к демоплате Pinboard [1]. Там стоит мой диспетчер очереди, а также немного автоматики на прерываниях. Плюс старая библиотека для HD44780

В принципе все с виду работает пучком и никаких косяков. Но одолели меня сомнения, так ли это? Если двигатель вращается, то это еще не значит, что он работает хорошо и без ошибок. Так и у нас, надо проверить, все ли в порядке. Слушать мы будем осциллографом «механику» тиканья службы таймера. Почему его? Ну так вокруг него вся логика программы построена.

Переназначаем выводы PD4 и PD5 на выход и будем их использовать для своих грязных целей. Для отладки в смысле.

Поставим дрыг импульсом на прерывание таймерной службы и посмотрим насколько красиво и правильно тикает системный таймер. Дискретность которого 1мс. И который определяет почти все выдержки в программе. В общем, прописываем в таймерное прерывание такую байду:

```
1 OutComp2Int:    SBI             BTA_P,BTB               ; For Debug
2
3                 TimerService                            ; Служба таймера диспетчера
4
5                 CBI             BTA_P,BTB               ; For Debug
6                 RETI
```

Вначале мы ставим бит порта PD4, а перед выходом сбрасываем. Это даст нам время выполнения и частоту выполнения. И тыкаемся нашим осциллографом, глядим:

Обана, а в таймере то глюки есть! Тикать то он тикает и с первого взгляда все нормально. Но почему то у него возникла аритмия и если это сейчас не вылезло, то может вылезти потом, при добавлении новых фич, где от четкости таймера будет многое зависеть. Причем глюк по времени плавающий, возникающий раз в сколько то сработок. А это значит, мы его трассировкой не поймаем. Можно, конечно, и аналитически протупить в код и найти, но это еще надо знать, что глюк есть. Да и код может быть такой, что сам черт ногу сломит.

Попробуем вычислить кто это нам тут таймер сбивает. Первое что приходит на ум — другое прерывание. Оно во время своей обработки запрещает другие прерывания и может сбить нам таймер. Что там у нас еще есть? Да хотя бы прерывание от АЦП. Выведем как его на вторую отладочную ногу:

```
1  ADC_INT:       SBI             BTA_P,BTC               ; For Debug
2
3                 PUSH            OSRG                    ; Сохраняем рабчий регистр
4                 IN              OSRG,ADCH               ; Берем данные из АЦП
5                 STS             ADC_Data,OSRG           ; Перекладываем их в ОЗУ
6                 POP             OSRG                    ; Восстанавливаем рабочий регистр
7
8                 CBI             BTA_P,BTC               ; For Debug
9
10                RETI                                    ; Выход из прерывания
```

Это будет дрыганье ногой PD5.

Смотрим что получилось:

То же самое, но на аналоговом осциллографе. Там не так наглядно, поэтому не стал выносить в пост и перегружать страницу [2]

Как видно из видео, у нас есть проблема — где то в коде есть процедура которая блокирует прерывания. Причем это не атомарный доступ, слишком уж большие задержки. Скорей что то похожее на цикл ожидания, в котором есть CLI/SEI конструкция. Где то что то я забыл удалить после отладки. Начинаем проглядывать код уже на предмет забытых прерываний. Мелкие процедурки видны сразу, а крупные, состоящие из нескольких файлов и макросов можно и прощупать.

Давайте как пощупаем задачу вывода на экран, переместим дрыгалку туда:

```
;-----------------------------------------------------------------------------
; Задача обновления экрана дисплея. Дисплей обновляется 5 раз в секунду, каждые 200мс
; данные берутся из видеопамяти и фоном записываются в контроллер HD44780
Fill:           SBI     BTA_P,BTC               ; ForDebug

                SetTimerTask    TS_Fill,200     ; Самозацикливание задачи через диспетчер таймеров

                LCDCLR                          ; Очистка дисплея

                LDZ     TextLine1               ; Взять в индекс Z адрес начала видеопамяти

                LDI     Counter,DWIDTH          ; Загрузить счетчик числом символов в строке

Filling1:       LD      OSRG,Z+                 ; Брать последовательно байты из видеопамяти, увеличивая индекс Z
                RCALL   DATA_WR                 ; И передавать их дисплею

                DEC     Counter                 ; Уменьшить счетчик.
                BRNE    Filling1                ; Пока не будет 0 (строка не кончилась) - повторять

                LCD_COORD 0,1                   ; Как кончится строка - перевести строку в дисплее

                LDI     Counter,DWIDTH          ; Опять загрузить длинной строки

Filling2:       LD      OSRG,Z+                 ; Адрес второй строки видеопамяти указывать не надо - они идут друг за другом
                RCALL   DATA_WR                 ; И таким же макаром залить вторую строку из видеопамяти

                DEC     Counter                 ; Уменьшаем счетчик
                BRNE    Filling2                ; Если строка не кончилась - продолжаем.

                CBI     BTA_P,BTC               ; ForDebug
                RET
```

И смотрим что получилось:

Аналоговый вариант данного действа. [3]

Как видим, у нас в задаче Fill есть запрет прерываний, который на значительный период блокирует не только таймерную службу, но и прерывания от АЦП и все остальное. При этом у нас один тик таймерной службы откладывается до тех пор, пока прерывания не разрешат. ВЫполняется, а в этом время уже нащелкало на второй тик и прерывание выполняется еще раз. Пока катастрофы не случилось, но будь Fill подольше, то мы бы прозевали один тик. А какое то событие уплыло по времени.

Это черевато тем, что разные синхронизированные по времени задачи могут глючить, причем глючить не всегда, а только тогда, когда на них выпадает совпадение по времени с Fill. Попробуй такое отловить. Тем более по

дефолту то считаем, что Fill работает и не глючит! Ведь раньше то ничего не мешало! Ну ну. Просто условия не создавались нужные.

Лезем в lcd4.asm и находим там запреты прерываний почти везде:

```
1
2
3
4
5
6
7
8
9 ;=============================================================================
1 =
0 BusyWait:       CLI                           ; Ожидание флага занятости контроллера
1 дисплея
1                 RCALL   PortIn                ; Порты на вход
1
2                 CBI     CMD_PORT,RS           ; Идет Команда!
1                 SBI     CMD_PORT,RW           ; Чтение!
3
1
4 BusyLoop:       SBI     CMD_PORT,E            ; Поднять строб
1                 RCALL   LCD_Delay             ; Подождать
5
1                 IN      R16,DATA_PIN          ; Считать байт
6
1                 PUSH    R16                   ; Сохранить его в стек. Дело в том, что у
7 нас R16 убивается в LCD_Delay
1
8                 CBI     CMD_PORT,E            ; Бросить строб - первый цикл (старший
1 полубайт)
9                 RCALL   LCD_Delay             ; Подождем маленько
2
0                 SBI     CMD_PORT,E            ; Поднимем строб
2                 RCALL   LCD_Delay             ; Подождем
1                 CBI     CMD_PORT,E            ; Опустим строб- нужно для пропуска второго
2 полубайта
2
2                 RCALL   LCD_Delay             ; Задержка снова
3
2                 POP     R16                   ; А теперь достаем сныканый байт - в нем
4 наш флаг. Может быть.
2
5                 ANDI    R16,0x80              ; Продавливаем по маске. Есть флаг?
2                 BRNE    BusyLoop              ; Если нет, то переход
6
2 BusyNo:         SEI                           ; Разрешаем прерывания.
7                 RET
2
8
2
9
3
0
3
1
```

И это только в процедуре ожидания флага готовности дисплея. Дальше оно есть и в процедуре чтения, записи и по всей библиотеке. Самое страшное, конечно, это в BusyWait т.к. оно зацикливается. Поэтому то и на половине задачи прерывания и срубались.

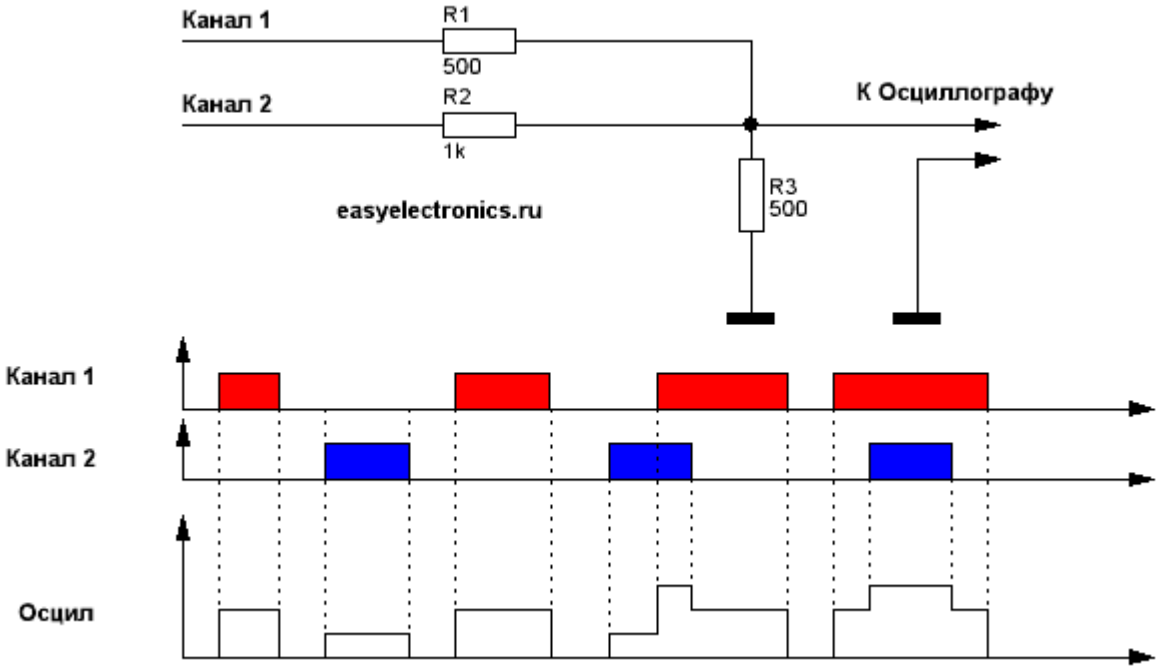
Е..й стыд! И я когда то это написал!? Нет, тогда запрет прерываний был оправдан. Я уже не помню почему, но без него в том случае было никак. Но вот переносить эти CLI/SEI в универсальную библиотеку это был полный маразм. Кстати, многие кто ее юзал отметили эту фишку с ненужным запретом прерываний. Многие, но далеко не все. Колитесь, кто не заглядывал в код и подключиле его к своей программе «as is» поленившись проверять и поверив на слово? ;) Все руки не доходят поправить ;)

Правим, заливаем в демоплату новую прошивку. Наблюдаем эффект:

Ничего не сломалось, все отлично работает. Done!

Ну и, напоследок, маленький трюк. Я тут вовсю юзаю многоканальный осцилл, а у меня тут еще и 16ти канальный анализатор есть… В общем, полный набор удовольствий. А кто то мучается с одноканальным осциллографом и как тут ему быть?

Не беда — можно применить одну хитрость. Обьединить несколько сигналов в один канал. Делается это вот по такой схеме:



Работает просто — когда на одном входе единичка, то напряжение с нее течет в землю. Этот ток создает падение напряжения на R3 которое видно на осциллографе. Когда на входе две единички, то через R3 уже идет ток с двух источников и падение напряжения становится суммой двух сигналов и это явно видно. Значения резисторов R1 и R2 лучше подбирать так, чтобы они были в пределах одного порядка, но различались раза в полтора-два. Это даст визуальное разделение сигналов и можно будет понять что к чему.

У меня же на демоплате я решил заюзать те резисторы, что уже стояли — 500омные на ограничение тока для светодиодов. А в качестве суммирующего применил один из переменников, выкрутив его на сопротивление около 800Ом. И с него отправил сигнал на осциллографы.
Резисторы там правда правда одинаковые и сигналы будут равные по высоте, но тут у нас по частоте уже понятно кто есть кто, так что не запутаемся :)

Вот что получилось:

Таким образом, можно на каждый канал загнать столько сигналов, что в экран не влезут. Правда анализировать их на глазок будет уже сложней. Но где наша не пропадала! :)

Игры с синхронизацией
Также не стоит забывать про такой канал как внешняя синхронизация (Ext Trig) Обычно на него забивают т.к. не

находят ему применения. А зря! Например, можно на него вывести какой либо сигнал который тоже надо ловить. И настроить осцил в такой режим, что без этого сигнала триггер не срабатывает (Normal режим, не Auto). И тогда если сигнала нет, то мы ничего и не увидим. Отсутствие информации тоже информация — о том, что сигнала нет :) А если есть, то у нас останутся еще два свободных канала на отлов последующих событий.

Также вовсю стоит играться с разными видами синхры. Аналоговые и самые дешевые цифровые ограничены только синхрой по фронтам, да всяким телесигналам. А вот более продвинутые цифровые, вроде ATTEN или тот же RIGOL (про Лекрои и прочие Тектроники я даже не вспоминаю) умеют синхронизироваться и по длине импульса, по расстоянию между импульсами ,по скорости нарастания/спада сигнала и еще по полудесятку разных параметров. Ловить ими сбои в работе программ милое дело! Раз уж раззорились на цифроосцил, так юзайте его возможности на все сто! :)))

Исходник программы на которой я баловался [4]

AVR. Учебный курс. Процедура сканирования клавиатуры

Итак, клавиатуру я сделал и написал процедуру сканирующую **клавиатурную матрицу 4x4 кнопки** [1]. Пора бы рассказать как организовать опрос такой клавы. Напомню, что клава представляет из себя строки, висящие на портах и столбцы, которые сканируются другим портом. Код написан для контроллера **ATMega8535**, но благодаря тому, что все там указано в виде макросов его можно быстро портировать под любой другой контроллер класса **Mega**, а также под большую часть современных **Tiny**. Хотя в случае с **Tiny** может быть некоторый затык ввиду неполного набора команд у них. Придется чуток дорабатывать напильником.
Короче, ближе к коду. Сразу оговорюсь, что я взял моду крошить один проект на десяток мелких файлов, а потом подключать их по мере необходимости. Во-первых, это резко структурирует код, позволяя легче в нем ориентироваться, а во-вторых, код становится модульным и его куски можно использовать как готовые библиотеки в других программах. Только подправить чуток. По этой же причине я все определения делаю через макросы, чтобы не пришлось править весь код, а достаточно было только пару строк изменить в файле конфигурации.

Теперь коротко о файлах:
keyboard_define.inc — файл конфигурации клавиатуры.
В этом файле хранятся все макроопределения используемые клавиатурой. Здесь мы задаем какие ножки микроконтроллера к какой линии подключены. Одна тонкость — выводы на столбцы (**сканирующий порт**) должны быть последовательным набором линий одного порта. То есть, например, ножки **0,1,2,3** или **4,5,6,7**, или **3,4,5,6**. Неважно какого порта, главное чтобы последовательно.
С определением ножек, думаю проблем не возникнет, а вот по поводу параметра **KEYMASK** я хочу рассказать особо.
Это маска по которой будет выделяться сканируемый порт. В ней должны быть 6 единиц и один 0. Ноль выставляется в крайне правую позицию сканирующего порта.

Пример:
У меня сканирующий порт висит на битах 7,6,5,4 крайне правый бит сканирующего порта это бит 4, следовательно маска равна 0b11101111 — ноль стоит на 4й позиции. Если сканирующие линии будут висеть на ножках 5,4,3,2, то маска уже будет 0b11111011 — ноль на второй позиции. Зачем это все будет объяснено ниже.

Также есть маска активных линий сканирующего порта — **SCANMSK**. В ней единицы стоят только напротив линий столбцов. У меня столбцы заведены на старшую тетраду порта, поэтому сканирующая маска имеет вид **0b11110000**.

В разделе инициализации нужно не забыть настроить ножки сканирующего порта на выход, а ноги считывающего на вход с подтяжкой. А потом вставить код обработчика клавиатуры куда-нибудь в виде обычной подпрограммы. Пользоваться просто — вызываем подпрограмму чтения с клавы, а когда возвращаемся у нас в регистре R16 находится скан код клавиши.

Вот так у меня выглядел тестовый код:

```
Main:    SEI                      ; Разрешаем прерывания.


         RCALL    KeyScan         ; Сканируем клавиатуру
```

```
        CPI     R16,0           ; Если вернулся 0 значит нажатия не было

        BREQ    Main            ; В этом случае переход на начало

        RCALL   CodeGen         ; Если вернулся скан код, то переводим его в

                                ; ASCII код.


        MOV     R17,R16         ; Загружаем в приемный регистр LCD обработчика

        RCALL   DATA_WR         ; Выводим на дисплей.


        RJMP    Main            ; Зацикливаем все нафиг.
```

Про **LCD** дисплей я пока ничего не скажу, так как процедуры еще не доведены до ума, но будут выложены и разжеваны в ближайшее время.

Теперь расскажу как работает **процедура KeyScan**

```
                .def    COUNT = R18

KeyScan: LDI    COUNT,4         ; Сканим 4 колонки

                LDI     R16,KEYMASK     ; Загружаем маску на скан 0 колонки.
```

Вначале мы подготавливаем сканирующую маску. Дело в том, что мы не можем вот так взять и гнать данные в порт. Ведь строки висят только на последних четырех битах, а на первых может быть что угодно, поэтому нам главное ни при каких условиях не изменить состояние битов младшей тетрады порта.

```
KeyLoop: IN     R17,COL_PORT    ; Берем из порта прежнее значение

                ORI     R17,SCANMSK     ; Выставляем в 1 биты сканируемой части.
```

Вначале **загружаем данные из регистра порта**, чтобы иметь на руках первоначальную конфигурацию порта. Также нам нужно выставить все сканирующие биты порта в 1, это делается посредством операции **ИЛИ** по сканирующей маске. В той части где стояли единицы после операции **ИЛИ** по маске **11110000** (мое значение **SCANMASK**) все биты станут единицами, а где был ноль останутся без изменений.

```
        AND     R17,R16         ; Сбрасываем бит сканируемого столбца

        OUT     COL_PORT,R17    ; Выводим сформированный байт из порта.
```

Теперь мы на сформированный байт накладываем **маску активного столбца**. В ней вначале ноль на первой позиции, а все остальные единицы. В результате, другие значения порта не изменятся, а вот в первом столбце возникнет 0. Потом маска сдвинется, а вся операция повторится снова. В результате ноль будет уже в следующем столбце и так далее. Таким образом, мы организуем «бегающий» нолик в сканирующем порте, при неизменности других, посторонних, битов порта. А дальше сформированное число загружается в регистр порта и ножки принимают соответствующие уровни напряжений.

```
        NOP     ; Задержка на переключение ноги.

        NOP

        NOP

        NOP
```

```
        SBIS    ROW0_PIN,ROW0    ; Проверяем на какой строке нажата

        RJMP    bt0


        SBIS    ROW1_PIN,ROW1

        RJMP    bt1


        SBIS    ROW2_PIN,ROW2

        RJMP    bt2


        SBIS    ROW3_PIN,ROW3

        RJMP    bt3
```

Серия **NOP** нужна для того, чтобы перед проверкой дать ножке время на то, чтобы занять нужный уровень. Дело в том, что реальная цепь имеет некоторое значение емкости и индуктивности, которое делает **невозможным мгновенное изменение уровня**, небольшая задержка все же есть. А на скорости в 8Мгц и выше процессор щелкает команды с такой скоростью, что напряжение на ноге еще не спало, а мы уже проверяем состояние вывода. Вот я и влепил несколько пустых операций. На 8Мгц все работает отлично. На большую частоту, наверное, надо будет поставить еще штук пять шесть **NOP** или влепить простенький цикл. Впрочем, тут надо поглядеть на то, что по байтам будет экономичней.
После циклов идет четыре проверки на строки. И переход на соответствующую обработку события.

```
        ROL     R16             ; Сдвигаем маску сканирования

        DEC     COUNT           ; Уменьшаем счетчик столбцов

        BRNE    KeyLoop         ; Если еще не все перебрали делаем еще одну итерацию


        CLR     R16             ; Если нажатий не было возвращаем 0


        RET


        .undef  COUNT
```

Вот тут происходит сдвиг маски влево командой циклического сдвига **ROL**. После чего мы уменьшаем счетчик итераций (изначально равен четырем, так как у нас четыре столбца). Если нажатий не было, то по окончании всех четырех итераций мы вываливаемся из цикла, обнуляем регистр **R16** и возвращаемся.

```
bt0:    ANDI    R16,SCANMSK             ; Формируем скан код

        ORI     R16,0×01        ; Возвращаем его в регистре 16

        RET
```

А вот один из возможных концов при нажатии. Тут формируется скан код который вернется в регистре **R16.** Я решил не заморачиваться, а как всегда зажать десяток байт и сделать как можно быстрей и короче. Итак, что мы имеем по приходу в этот кусок кода. А имеем мы один из вариантов сканирующего порта (**1110,1101,1011,0111**), а также знаем номер строки по которой мы попали сюда. Конкретно в этот кусок можно

попасть только из первой строки по команде **RJMP bt0.**
Так давай сделаем скан код из сканирующей комбинации и номера строки! Сказано — сделано! Сначала нам надо выделить из значения порта сканирующую комбинацию — она у нас хранится в регистре **R16**, поэтому выковыривать из порта ее нет нужды. Продавливаем операцией И значение **R16** через **SCANMASK** и все что было под единичками прошло без изменений, а где были нули — занулилось. Опа, и у нас выведен сканирующий кусок — старший полубайт. Теперь вклеим туда номер строки — операцией **ИЛИ**. Раз, и получили конструкцию вида **[скан][строка]**
Вот ее и оставляем в регистре **R16**, а сами выходим прочь! Также и с остальными строками. Погляди в исходнике, я их не буду тут дублировать.

Декодирование скан кода.
Отлично, скан код есть, но что с ним делать? Его же никуда не приткнуть. Мы то знаем, что вот эта шняга вида **01110001** это код единички, а какой нибудь **LCD** экран или стандартная терминалка скорчит нам жуткую кракозябру и скажет, нам все что она думает о нашей системе обозначений — ей видите ли **ASCII** подавай. Ладно, будет ей ASCII.

Как быть? Прогнать всю конструкцию по **CASE** где на каждый скан код присвоить по **ASCII** коду меня давит жаба — это же сколько надо проверок сделать! Это же сколько байт уйдет на всю эту тряхомудию? А память у нас не резиновая, жалкие восемь килобайт, да по два байта на команду, это в лучшем случае. Я мог все это сделать прям в обработчике клавиатуры. НЕТ!!! В ТОПКУ!!! Мы пойдем своим путем.
Ок, а что у нас есть в запасе? Метод таблиц перехода не катит, по причине жуткой неупорядоченности скан кодов. Почесал я тыковку, пошарился по квартире… и тут меня осенило. Конечно же!!! **Брутфорс!!!**

Брутфорсим скан код.
Итак, у нас есть жутко несваримый скан код, а также стройная таблица **ASCII** символов. Как скрестить ужа с ежом? Да все просто! Разместим в памяти таблицу символов в связке **[скан код]:[ascii код]** , а потом каждый нужный скан код будем прогонять через эту таблицу и при совпадении подставлять на выходе нужный **ASCII** из связки. Классический пример программизма — потеряли во времени, зато выиграли в памяти.

Вот так это выглядит:

```
CodeGen:LDI     ZH,High(Code_Table*2)     ; Загрузил адрес кодовой таблицы

        LDI     ZL,Low(Code_Table*2)      ; Старший и младший байты
```

Тут мы загрузили в индексный регистр адрес нашей таблицы. Умножение на два для того, чтобы адрес был в байтах, т.к. в среде компилятора пространство кода адресуется в словах.

```
Brute:  LPM     R17,Z+          ; Взял из таблицы первый символ — скан код


        CPI     R17,0xFF ; Если конец таблицы


        BREQ    CG_Exit         ; То выходим



        CPI     R16,0           ; Если ноль,


        BREQ    CG_Exit         ; то выходим



        CP      R16,R17         ; Сравнил его со скан кодом клавиши.


        BREQ    Equal           ; Если равен, то идем подставлять ascii код
```

Загружаем из таблицы первый скан код и нычим его в регистр **R17**, попутно увеличиваем адрес в регистре **Z** (выбор следующей ячейки таблицы) и первым делом сравниваем его с **FF** — это код конца таблицы. Если таблица закончилась, то выходим отсюда. Если мы не всю таблицу перебрали, то начинаем сравнивать входное значение

(в регистре **R16**) вначале с нулем (нет нажатия), если ноль тоже выходим. И со скан кодом из таблицы. Если скан таблицы совпадает со сканом на входе, то переходим на **Equal**.

```
        LPM     R17,Z+          ; Увеличиваем Z на 1

        RJMP    Brute           ; Повтор цикла
```

А в случае если ничо не обнаружено, то мы повторно вызываем команду **LPM R17,Z+** лишь для того, чтобы она увеличила **Z** на единичку — нам же надо перешагнуть через **ASCII** код и взять следующий скан код из таблицы. Просто **INC Z** не прокатит, так как **Z** у нас **двубайтный**. **ZL и ZH**. В некторых случаях достаточно **INC ZL**, но это в случае когда мы точно уверены в том, что адрес находится недалеко от начала и переполнения младшего байта не произойдет (иначе мы вместо адреса 00000001:00000000 получим просто 00000000:0000000, что в корне неверно), а команда **LPM** все сделает за нас, так что тут мы сэкономили еще пару байт. Потом мы вернемся в начало цикла, а там будет опять **LPM** которая загрузит уже следующий скан код.

```
Equal:  LPM     R16,Z           ; Загружаем из памяти ASCII код.

        RET                     ; Возвращаемся
```

Если же было совпадение, то в результате **LPM Z+** у нас **Z** указывает на следующую ячейку — с **ASCII** кодом. Ее мы и загружаем в регистр **R16** и выходим наружу.

```
CG_Exit: CLR    R16     ; Сбрасываем 0 = возвращаем 0

        RET                     ; Возвращаемся
```

А в случае нулевого исхода, когда либо таблица кончилась, а скан код так и не подобрался, либо ноль был в регистре R16 на входе — возвращаемся с тем же нулем на выходе. Вот так вот.

```
;=======================================

; STATIC DATA

;=======================================

Code_Table:     .db     0×71,0×31       ;1

                .db     0xB1,0×32       ;2

                .db     0xD1,0×33       ;3

                .db     0×72,0×34       ;4

                .db     0xB2,0×35       ;5

                .db     0xD2,0×36       ;6

                .db     0×73,0×37       ;7

                .db     0xB3,0×38       ;8

                .db     0xD3,0×39       ;9

                .db     0×74,0×30       ;0

                .db     0xFF,0          ;END
```

Тут просто табличка статичных данных, на границе памяти. Как видишь данные сгруппированы по два байта — сканкод/ASCII

Вот посредством таких извратов вся программа, с обработкой клавиатуры, декодированием скан кода, чтением/записью в LCD индикатор и обнулением оперативки (нужно для того, чтобы точно быть уверенным, что память равна нулю) заняло **всего 354 байта**. Кто сможет меньше?

- **Архив со всеми исходными файлами и проектом в AVR Studio** [2]

AVR. Учебный курс. Подключение к AVR LCD дисплея HD44780

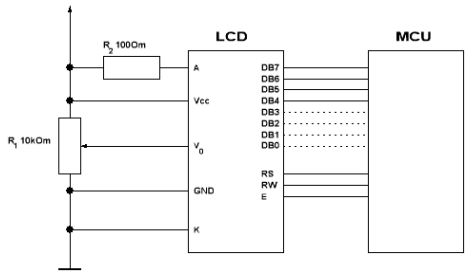
Так случилось, что прикупил я тут себе поприколу **LCD** дисплейчик две строки по восемь символов. Валялся он в ящике валялся, да чегото поперло меня и решил я его заюзать, попутно вкурив в его работу. О том как подключить к **AVR LCD** дисплей я вам сейчас и поведаю.

Для начала оговорюсь сразу, что речь тут пойдет о **LCD** индикаторах на контроллере **HD44780,** который стал промышленным стандартом де-факто на рынке цифро-буквенных дисплеев. Продается везде где только можно, стоит недорого (8x2 мне обошелся порядка 150 рублей), а также под него написана куча кода. Я же, как обычно, решил изобрести велосипед и сварганить свою собственную тру-библиотеку для работы с этим типом индикаторов. Разумеется на ассемблере, а на чем же еще? ;)

Подключение.
LCD на базе **HD44780** подключается к **AVR** микроконтроллеру напрямую к портам. Есть два способа подключения — на 8 бит и на 4 бита. В восьмибитном режиме немножко проще закидывать байты — не нужно сдвигать байт, зато в четырех битном резко нужно тратить на целых четыре ножки контроллера меньше. Есть еще одна особенность работы в 8-битном режиме — к некоторым контроллерам можно подрубить этот дисплей **как внешнее ОЗУ** и засылать данные простыми командами пересылки. Лично я подключил его в режиме полного порта у меня один фиг выводы уже девать некуда было, так что не жалко.

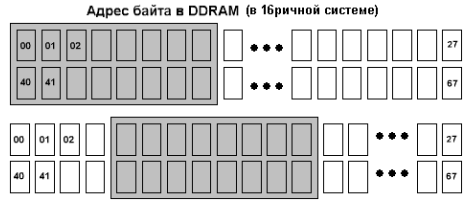
- Выводы **DB7...DB0** это шина данных/адреса.
- **E** — стробирующий вход. Дрыгом напряжения на этой линии мы даем понять дисплею что нужно забирать/отдавать данные с/на шину данных.
- **RW** — определяет в каком направлении у нас движутся данные. Если 1 — то на чтение из дисплея, если 0 то на запись в дисплей.
- **RS** — определяет что у нас передается, команда (RS=0) или данные (RS=1). Данные будут записаны в память по текущему адресу, а команда исполнена контроллером.

Со стороны питания все еще проще:



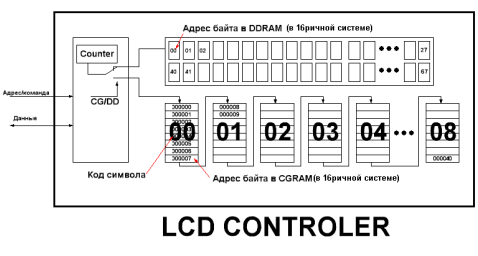
[2]

Подключение дисплея



[3]

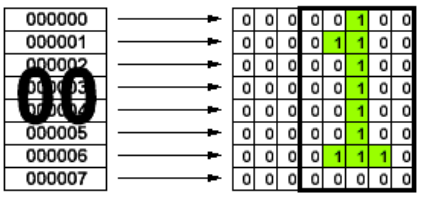
Видимая и скрытая область экранной памяти



[4]

Структура адресации контроллера

HD44780



[5]

Формирование символа в ячейке CGRAM

- **GND** — минус, он же общий.
- **Vcc** — плюс питания, обычно 5V
- **V0** — вход контрастности. Сюда нужно подавать напряжение от нуля до напряжения питания, тем самым задается контрастность изображения. Можно поставить переменный резистор, включенный потенциометром и крутить в свое удовольствие. Главное поймать значение максимального контраста, но чтобы не было видно знакомест (серый ореол из квадратов вокруг символа). Если же выставить слишком малый контраст, то символы будут переключаться лениво и задумчиво. Примерно как в калькуляторе у которого сели батарейки.
- **A** — это вход Анода светодиодной подсветки. Короче плюс.
- **K** — соответственно Катод, он же минус. Подсветка хавает примерно 100мА и поэтому нужно выставить туда токоограничительный резистор на 100 Ом. Кстати, многие ЖК дисплеи имеют на плате пятачки для припайки резисторов. Если прозвонить, то можно убедиться в том, что эти линии ведут на входы питания LCD, поэтому, впаяв резисторы, можно не заморачиваться на запитку подстветки, она будет подключена к питанию контроллера.

Логическая структура LCD контроллера HD44780

Контроллер имеет свой блок управления, который обрабатывает команды и память. Она делится на три вида:

DDRAM — память дисплея. Все что запишется в **DDRAM** будет выведено на экран. То есть, например, записали мы туда код **0×31** — на экране выскочит символ **«1”** т.к. **0x31** это ASCII код цифры **1.** Но есть тут одна особенность — **DDRAM** память гораздо больше чем видимая область экрана. Как правило, **DDRAM** содержит **80 ячеек** — 40 в первой строке и 40 во второй, а на дисплей может двигаться по этой линейке как окошко на логарифмической линейке, высвечивая видимую область. То есть, например, можно засунуть в **DDRAM** сразу пять пунктов меню, а потом просто гонять дисплей туда сюда, показывая по одному пункту. Для перемещения дисплея есть спец команда. Также есть понятие курсора — это место в которое будет записан следующий символ, т.е. текущее значение счетчика адреса. Курсор не обязательно может быть на экране, он может располагаться и за экраном или быть отключен вовсе.

CGROM — таблица символов. Когда мы записываем в ячейку **DDRAM** байт, то из таблицы берется символ и рисуется на экране. **CGROM** нельзя изменить, поэтому важно, чтобы она имела на борту русские буквы. Если, конечно, планируется русскоязычный интерфейс.

CGRAM — тоже таблица символов, но ее мы можем менять, создавая свои символы. Адресуется она линейно, то есть вначале идет 8 байт одного символа, построчно, снизу вверх — один бит равен одной точке на экране. Потом второй символ тем же макаром. Поскольку знакоместо у нас 5 на 8 точек, то **старшие три бита роли не играют**. Всего в **CGRAM** может быть 8 символов, соответственно **CGRAM** имеет **64** байта памяти. Эти программируемые символы имеют коды от 0x00 до 0x07. Так что, закинув, например, в первые **8 байт CGRAM** (первый символ с кодом 00) какую нибудь фигню, и записав в **DDRAM** нуль (код первого символа в **CGRAM**) мы увидим на экране нашу хрень.

Доступ к памяти.
Тут все просто. Мы командой выбираем в какую именно память и начиная с какого адреса будем писать. А потом просто шлем байты. Если указано, что записываем в DDRAM то на экран (или в скрытую область) полезут символы, если в CGRAM то байты полезут уже в память знакогенератора. Главное потом не забыть переключится обратно на область **DDRAM**

Система команд.
Система команд проста как мычание. О том, что передается команда контроллеру дисплея сообщит нога **RS**=0. Сама команда состоит из старшего бита, определяющего за что отвечает данная команда и битов параметров, указывающих контроллеру HD44780 как дальше жить.

Таблица команд:

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Значение

0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	Очистка экрана. Счетчик адреса на 0 позицию DDRAM
0	**0**	**0**	**0**	**0**	**0**	**1**	-	Адресация на DDRAM сброс сдвигов, Счетчик адреса на 0
0	**0**	**0**	**0**	**0**	**1**	I/D	S	Настройка сдвига экрана и курсора
0	**0**	**0**	**0**	**1**	D	C	B	Настройка режима отображения
0	**0**	**0**	**1**	S/C	R/L	-	-	Сдвиг курсора или экрана, в зависимости от битов
0	**0**	**1**	DL	N	F	-	-	Выбор числа линий, ширины шины и размера символа
0	**1**	AG	AG	AG	AG	AG	AG	Переключить адресацию на SGRAM и задать адрес в SGRAM
1	AD	AD	AD	AD	AD	AD	AD	Переключить адресацию на DDRAM и задать адрес в DDRAM

Теперь поясню что значат отдельные биты:

- **I/D** — инкремент или декремент счетчика адреса. По дефолту стоит 0 — Декремент. Т.е. каждый следующий байт будет записан в n-1 ячейку. Если поставить 1 — будет Инкремент.
- **S** - сдвиг экрана, если поставить 1 то с каждым новым символом будет сдвигаться окно экрана, пока не достигнет конца DDRAM, наверное удобно будет когда выводишь на экран здоровенную строку, на все 40 символов, чтобы не убегала за экран.
- **D** — включить дисплей. Если поставить туда 0 то изображение исчезнет, а мы в это время можем в видеопамяти творить всякие непотребства и они не будут мозолить глаза. А чтобы картинка появилась в эту позицию надо записать 1.
- **C** - включить курсор в виде прочерка. Все просто, записали сюда 1 — включился курсор.
- **B** — сделать курсор в виде мигающего черного квадрата.
- **S/C** сдвиг курсора или экрана. Если стоит 0, то сдвигается курсор. Если 1, то экран. По одному разу за команду
- **R/L** — определяет направление сдвига курсора и экрана. 0 — влево, 1 — вправо.
- **D/L** — бит определяющий ширину шины данных. 1-8 бит, 0-4 бита
- **N** — число строк. 0 — одна строка, 1 — две строки.
- **F** - размер символа 0 — 5x8 точек. 1 — 5x10 точек (встречается крайне редко)
- **AG** - адрес в памяти **CGRAM**
- **AD** — адрес в памяти **DDRAM**

Я сам долго тупил в эту табличку, пытаясь понять, что же от меня хотят. Видимо был невыспавшийся, но и вправду, она на первый взгляд не очевидна, поэтому подкреплю все примером.

Задача:

1. Включить дисплей.
2. Очистить содержимое.
3. Сдвинуть курсор на одну позицию.
4. И записать туда «1”.

Решение (последовательность команд):

Первым делом **Инициализация** дисплея без которой большая часть дисплеев на HD44780 просто откажется работать. Некоторые виды имеют дефолтные состояние (шина 8 бит, курсор в 0) и им только дисплей включить. Но все же ее лучше сделать, мало ли что там намудрил разработчик. Лишней не будет.

1. **001**11000 Шина 8 бит, 2 строки
2. **00000001** Очистка экрана
3. **00000**110 Инкремент адреса. Экран не движется

1. **00001**100 Включили дисплей (D=1)
2. **00000001** Очистили дисплей. Указатель встал на DDRAM

3. **0001**0100 Сдвинули курсор (S/C=0) вправо (R/L=1)
4. 00110001 — это мы уже записали данные (ножка RS=1) код «1” 0x31

Жирным шрифтом выделен идентификатор команды, ну а остальное по таблице увидите.

Задача: создать свой символ. С кодом 01 и вывести его на экран.
Считаем, что дисплей у нас уже инициализирован и готов к приему данных.

Решение:

1. **01**001000 Выбираем в **CGRAM** адрес 0x08 — как раз начало второго символа (напомню, что на один символ уходит 8 байт)
2. 00000001 Это пошли 8 байт данных. (**RS=1**)
3. 00000010 Рисуем значок молнии, ну или
4. 00001000 ССовскую Зиг руну, кому как
5. 00010000 больше нравится.
6. 00011111 Старшие три бита не действуют
7. 00000010 Туда можно писать что угодно, на
8. 00001000 результат влиять не будет.
9. 00010000 Последний байт данных
10. 10000000 А это уже команда — переключение адреса на **DDRAM** и указатель на адрес **0000000** - первый символ в первой строке.
11. 00000001 И снова данные (**RS=1**), код 01 — именно в него мы засунули нашу молнию.

Опа и он на экране!

Так, с логикой разобрались, пора вкуривать в физику протокола общения. Код я приведу несколько позже, когда вылижу свою библиотеку и заоптимизирую до состояния идеала. Пока же дам алгоритм, а его уж на любом языке программирования реализовать можно. Хоть на ассемблере, хоть на Сях, да хоть на Васике :)

Алгоритм чтения/записи в LCD контроллер HD44780
Направление, а также команда/данные определяются ножками, а чтение и запись осуществляется по переходу строба (вывод E) из 1 в 0

Инициализация портов

1. RS, RW, E — в режим выхода.
2. DB7..DB0 в режим входа. Впрочем, можно их не трогать, дальше переопределим.

Ожидание готовности, чтение флага занятости.

1. Порт данных на вход с подтяжкой (DDR=0, PORT=1)
2. RS=0 (команда)
3. RW=1 (чтение)
4. E=1 (Готовьсь!!!)
5. Пауза (14 тактов процессора на 8МГЦ хватало)
6. E=0 (Пли!)
7. Читаем из порта. Если бит 7 (Busy flag) установлен, то повторяем все заново, пока не сбросится.

Запись команды

1. Ожидание готовности
2. RS=0 (команда)
3. RW=0 (запись)
4. E=1 (Готовьсь!!!)
5. Порт на выход
6. Вывести в порт код команды

7. Пауза
8. E=0 (Пли!)
9. ~~Орудие на плечо~~ Порт на вход, на всякий случай.

Запись Данных

1. Ожидание готовности
2. RS=1 (Данные)
3. RW=0 (запись)
4. E=1 (Готовьсь!!!)
5. Порт на выход
6. Вывести в порт код команды
7. Пауза
8. E=0 (Пли!)
9. Порт на вход, на всякий случай.

Чтение команды

1. Ожидание готовности
2. Порт данных на вход с подтяжкой (DDR=0, PORT=1)
3. RS=0 (команда)
4. RW=1 (чтение)
5. E = 1 (Готовьсь! В этот момент данные из LCD вылазят на шину)
6. Пауза
7. Считываем данные с порта
8. E=0 (Ать!)

Чтение Данных

1. Ожидание готовности
2. Порт данных на вход с подтяжкой (DDR=0, PORT=1)
3. RS=1 (Данные)
4. RW=1 (чтение)
5. E = 1 (Готовьсь! В этот момент данные из LCD вылазят на шину)
6. Пауза
7. Считываем данные с порта
8. E=0 (Ать!)

С четырех разрядной шиной все точно также, только там каждая операция чтения/записи делается за два дрыга строба.

Запись:

1. E=1
2. Пауза
3. Выставили в порт старшую тетраду
4. E=0
5. Пауза
6. E=1
7. Пауза
8. Выставили в порт младшую тетраду
9. E=0

Чтение

1. E=1
2. Пауза
3. Читаем из порта старшую тетраду
4. E=0
5. Пауза
6. E=1
7. Пауза
8. Читаем из порта младшую тетраду
9. E=0

Ждите код :) Скоро будет :)
UPD:
А вот и код! [6]

AVR. Учебный Курс. Библиотека для LCD на базе HD44780

Сел я и дописал свою **библиотеку для LCD на базе HD44780**.

Как она работает я тут расписывать не буду — код весьма плотно фарширован комментариями. Тем более я уже **рассказывал как работать с этим дисплеем** [1] Поэтому, думаю, разберетесь. Если будут вопросы, то обращайтесь. Тут же я расскажу как ей пользоваться.

Состав
Библиотека состоит из двух файлов **LCD.asm** и **LCD_macro.inc** для подключения по 8ми битной шине и **LCD4.asm** и **LCD4_macro.inc** для подключения по четырех битной шине данных. Используете тот вариант, по которому у вас подключен дисплей.

- Файл **LCD.asm** содержит все основные настройки портов и, собственно, код.
- **LCD_macro.inc** содержит макросы для работы с дисплеем. И используется для работы с библиотекой.

Подключение **LCD** к микроконтроллеру.
Порт данных использует биты **7...4 любого порта** на 4 битном подключении, или весь порт целиком на 8ми разрядном
Порт управления использует **3 любых бита любого порта**. Главное, чтобы они были **на одном порту**. Впрочем код можно и чуток подправить :)

Подключение библиотеки в код.
Где нибудь в начале программы, там где обычно определяют макросы, добавляете строчку

```
1 .include "LCD4_macro.inc"
```

А в конце кода, там где все процедуры, добавляете

```
1 .include "LCD4.asm"
```

Разумеется файл должен быть в папке проекта, иначе компилятор его не найдет.

Настройка
В файле **LCD.asm** есть раздел **LCD_Define**

```
1  ;=========== LCD Define ====================================
2          .equ    DATA_PORT       = PORTB         ; LCD Порт данных
```

```
3          .equ    DATA_PIN        = PINB
4          .equ    DATA_DDR        = DDRB
5
6          .equ    CMD_PORT        = PORTA         ; LCD Порт управления
7          .equ    CMD_PIN         = PINA
8          .equ    CMD_DDR = DDRA
9
10         .equ    E               = 0             ; Бит строба
11         .equ    RW              = 1             ; Бит чтения/записи
12         .equ    RS              = 2             ; Бит команда/Данные
13
14         .equ    SPEED   = 14    ; 14 для XTAL=16MHz, 10 для XTAL=8MHz,
15                                 ; 6 для XTAL=4MHz, 5 для XTAL<4MHz
```

Порт данных и порт управления указывай те, которые у тебя используются. У меня вот это B и A. Также не забудь указать к каким битам подключены линии управления.
Параметр **SPEED** задается в зависимости от скорости кристалла. Больше можно, меньше нет. Можно поставить 14 и не парится. Но некрасиво :)

В файле **LCD_macro.inc** нужно найти блок настроек дисплея и выставить там нужные биты. Выглядит это так:

```
1          .MACRO  INIT_LCD                                ; Инициализация LCD
2          RCALL   InitHW                                  ; Настроить контрольный порт
3          RCALL   LCD_DELAY                               ; Подождать
4          WR_CMD  (1<<LCD_F)|(0<<LCD_F_8B)                ; Выдать функцию смены разрядности.
5          WR_CMD  (1<<LCD_F)|(0<<LCD_F_8B)|(1<<LCD_F_2L)        ; Дважды.
6
7  ; Так как по дефолту шина 8 бит и нельзя передать сразу вторую половину байта.
8
9          WR_CMD  (1<<LCD_CLR)                                                    ;0x01
10         WR_CMD  (1<<LCD_ENTRY_MODE)|(1<<LCD_ENTRY_INC)                          ;0x06
11         WR_CMD  (1<<LCD_ON)|(1<<LCD_ON_DISPLAY)|(0<<LCD_ON_CURSOR)|(0<<LCD_ON_BLINK) ;0x0C
12         WR_CMD  (1<<LCD_HOME)
13         .ENDM
```

Перечень опций, сгруппированных по типу приведен там чуть выше. Обращу внимание на то, что инициализация 4х разрядной шины идет ДВА РАЗА. Первый раз мы говорим контроллеру, что у нас шина 4 разрядная. Но при этом мы не можем сказать второй полубайт слова управления. Поэтому, когда контроллер перейдет на 4х разрядный режим, выдаем нашу посылку снова, на этот раз вторая тетрада нормально пройдет.

Использование
Перед первым использованием надо сделать инициализацию индикатора. Делается это макросом

```
1          INIT_LCD
```

При этом задаются параметры дисплея, указанные в разделе инициализации.

Запись байта в дисплей осуществляется макросами

```
1          WR_CMD xx
2          WR_DATA xx
```

Где вместо **xx** вписываем наш байт.

Если надо выдать на экран, что либо программно сгенеренное, в таком случае используется прямой вызов процедуры. Байт передается **через регистр R17.**

```
1          RCALL   CMD_WR
2          RCALL   DATA_WR
```

Чтение осуществляется аналогичным образом:

```
1         RD_CMD
2         RD_DATA
```

или

```
1         RCALL   CMD_RD
2         RCALL   DATA_RD
```

В данном случае макрос и процедура ничем не отличаются. Сделано просто для унификации. Считанный байт будет в регистре **R17**

Установка координаты курсора осуществляется макросом
LCD_COORD X,Y
Где X и Y координаты по горизонтали, и вертикали в знакоместах.

Сдвиг курсора или экрана осуществляется макросом **SHIFT xx**, где вместо **xx** подставляются нужные аргументы

```
1         SHIFT SCR_L     ; сдвиг экрана влево
2         SHIFT SCR_K     ; сдвиг экрана вправо
3         SHIFT CUR_L     ; сдвиг курсора влево
4         SHIFT CUR_R     ; сдвиг курсора вправо
```

Макрос **LCDCLR** делает полную очистку видеопамяти.

Макрос **WR_CGADR xx** позволяет установить указатель в область памяти знакогенератора, чтобы можно было записать свой символ. Если затем начать слать данные, то они будут записаны в ячейки знакогенератора.
Макрос **RD_CGADR xx** позволяет установить указатель в область памяти знакогенератора,чтобы можно было считать байт из знакогенератора.
Макрос **WR_DDADR xx** устанавливает указатель на видео память. С этого момента все данные попадают на экран.

Для примера попробуем создать зиг руну, ранее упомянутую в записи про описании контроллера

```
1          INIT_LCD               ; Инициализируем
2
3          WR_CGADR 0             ; Указатель на начало знакогенератора.
4
5          WR_DATA 0b00000001     ; Запись данных нового знака
6          WR_DATA 0b00000010
7          WR_DATA 0b00000100
8          WR_DATA 0b00011111
9          WR_DATA 0b00000010
10         WR_DATA 0b00000100
11         WR_DATA 0b00001000
12
13         WR_DDADR 0             ; Указатель на начало видео памяти (ячейка с координатми
14 0,0)
15
           WR_DATA 0              ; У новоиспеченного символа код 0 напечатем его
```

Системные требования и ограничения
Используется система команда ATMega на некоторых ATTiny может начать ругаться. Придется править. Занимает 474 байта кода в максимальном исполнении (4 битная шина). На 8ми битной шине будет байт на 100 короче. Если выкинуть процедуру чтения, то будет еще короче.

Процедура активно использует регистр R16 и R17, поэтому их содержимое меняется непредсказуемым образом!!! Так что либо учитывайте это, либо сохраняйте заранее эти регистры.

- **Архив с библиотекой для LCD дисплея** [2]

Вот так вот, если что будет нового то выпущу версию 2. Пользуйтесь.

AVR. Учебный Курс. Виртуальные порты

Глядя на то, как раскиданы порой ножки портов по корпусу контроллеров, у меня возникают большое подозрение, что разводчик кристалла дунул что то сильно забористое. Когда вперемешку идут выводы разных портов, да еще почти в рандомном порядке… Когда к этим портам вешаем что-либо разнобойное, то пофигу. А если надо подцепиться на прямую линейку шины данных, вроде того же **LCD** дисплея? Вот тут и начинается круголяние дорожек по плате. А если плата фаршированная донельзя? Тут приходится вводить перемычки, дополнительные слои и извращаться как только можно. Короче, та еще проблема.

Я решаю ее **виртуальными портами**. То есть завожу в системе переменную каждому биту которой привязываю какую-либо ножку. Поскольку все выводы независимы я могу компоновать виртуальный порт в любом порядке и из любых линий. Чертовски удобно. Впрочем, за удобство приходится платить — обработка такого порта требует несколько десятков команд. Но когда кристалл не забит под завязку, то можно позволить себе немного пошиковать во имя удобства.

Итак. Подключаем куда-нибудь в область процедур блок кода виртуального порта.

```
1         .include        "VPort.asm"
```

Теперь надо сконфигурировать виртуальный порт. Все делается в **define** записи в файле **VPort.asm**

```
1  ;===================================
2  ; Virtual Port ==== Virtual Port
3  ;===================================
4  .def    VP_REG  = R16
5
6  .equ    VPort0  = PORTD
7  .equ    VPort1  = PORTD
8  .equ    VPort2  = PORTD
9  .equ    VPort3  = PORTD
10 .equ    VPort4  = PORTD
11 .equ    VPort5  = PORTC
12 .equ    VPort6  = PORTC
13 .equ    VPort7  = PORTC
14 .equ    VDDR0   = DDRD
15 .equ    VDDR1   = DDRD
16 .equ    VDDR2   = DDRD
17 .equ    VDDR3   = DDRD
18 .equ    VDDR4   = DDRD
19 .equ    VDDR5   = DDRC
20 .equ    VDDR6   = DDRC
21 .equ    VDDR7   = DDRC
22
23 .equ    VPIN0   = PIND
24 .equ    VPIN1   = PIND
25 .equ    VPIN2   = PIND
26 .equ    VPIN3   = PIND
27 .equ    VPIN4   = PIND
28 .equ    VPIN5   = PINC
29 .equ    VPIN6   = PINC
30 .equ    VPIN7   = PINC
31
32 .equ    VP0     = 3
33 .equ    VP1     = 4
34 .equ    VP2     = 5
```

```
35 .equ    VP3     = 6
36 .equ    VP4     = 7
37 .equ    VP5     = 0
38 .equ    VP6     = 1
39 .equ    VP7     = 2
```

Параметр **VP_REG** — указывает какой регистр мы будем юзать для обращения с портом. Данный регистр юзается **только для записи и чтения** из порта. Данные в нем **не хранятся**. Но можно сделать три регистра для **DDR**, **PORT** и **PIN**. Просто мне как то западло это было делать.
Также можно, по принципу процедуры **VRPIN** считывать по мере надобности данные и из **PORT** с **DDR**, но это потребует дофига памяти.

К парметрам **VPORTx, VDDRx** и **VPINx** привязываем конкретные регистры портов, а к параметру **VP** номер пина. В скопированном сюда куске кода у меня виртуальный порт состоит наполовину из порта C на половину из порта D.

```
1  VRPort: SBRS    VP_REG,0
2          CBI     VPORT0,VP0
3          SBRC    VP_REG,0
4          SBI     VPORT0,VP0
5
6  . . .
7          SBRS    VP_REG,7
8          CBI     VPORT7,VP7
9          SBRC    VP_REG,7
10         SBI     VPORT7,VP7
11         RET
12
13 VRDDR:  SBRS    VP_REG,0
14         CBI     VDDR0,VP0
15         SBRC    VP_REG,0
16         SBI     VDDR0,VP0
17
18 . . .
19
20         SBRS    VP_REG,7
21         CBI     VDDR7,VP7
22         SBRC    VP_REG,7
23         SBI     VDDR7,VP7
24         RET
25
26 VRPIN:  CLR     VP_REG
27         SBIC    VPIN0,VP0
28         SBR     VP_REG,1
29
30 . . .
31
32         SBIC    VPIN7,VP7
33         SBR     VP_REG,128
34         RET
```

Эти три процедуры выполняют запись в реальные DDR, PORT и чтение из PIN порта. Выдавая все это дело в регистр виртуального порта.
При записи используется двойная проверка — вначале проверяем есть ли бит — если есть, то выставляем. Потом проверяем на отстствие бита — если нет, то ставим. Данный метод позволяет менять состояние отдельных битов, не трогая состояние соседней и не дергая состояние, если оно не изменилось. Для PORT и DDR это критично, так как появление там не того уровня черевато выгоранием порта или непредсказуемыми последствиями для периферии. А вот с чтением PIN все куда проще. Вначале мы сбрасываем выходной регистр виртуального порта, а потом выставляем биты, в соответствии с наличии битов в реальных PIN ячейках. Команда SBR выставляет указаные биты. Фактически она аналогична команде ORI. Нафига была делать две разные команды одной длины, делающие одно и то же, за то же число тактов, я так и не догнал. ИМХО это один из AVR маразмов. Кои тут встречаются порой.

Использование
Загрузка данных в виртуальный PORT или DDR:

```
1         LDI     VP_REG,0xFF
2         RCALL   VRPort
3
4         LDI     VP_REG,0x00
5         RCALL   VRDDR
```

А чтение осуществляется одной командой:

```
1         RCALL   VRPIN
```

После чего данные достаются из регистра VR_REG

При реальном использовании процедуры эти желательно кромсать и обрезать. Например, срезая неиспользованые биты порта. Если дрыгание PORT и DDR не критично к импульсам смены состояния (скажем у клавиатуры). То можно сделать запись в VRDDR и VRPORT с предварительным обнулением, а потом с заносом единиц. Как это сделано в VRPIN.

Библиотека виртуального порта Vport.asm [1]

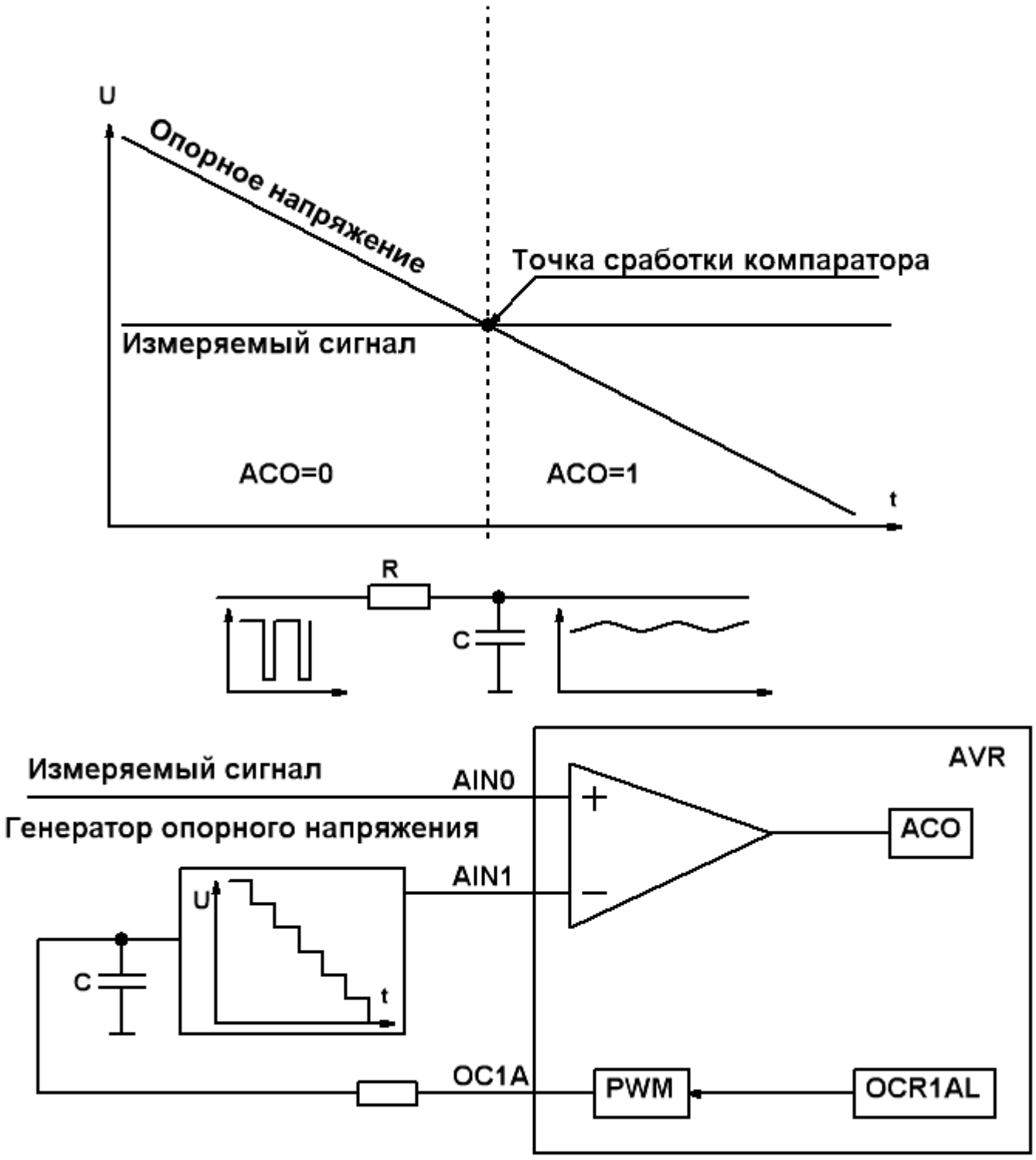
Вот такая вот загогулина. Пользуюсь довольно давно. Всем рекомендую — удобно.

AVR. Учебный курс. Делаем АЦП из Аналогового компаратора

Так сложилось, что основной МК с которым я работаю постоянно и на котором делаю подавляющее большинство задач это **ATTiny2313** — он популярен, а, главное, это **самый дешевый контроллер** из всей линейки **AVR** с числом ног более 8. Я их брал числом около трех сотен за 18, чтоль, рублей штучка. Но вот западло — **у него нет АЦП**. Совсем нет. А тут он понадобился — нужно замерить сигнал с датчика. Засада. Не переходить же из-за такой фигни на более фаршированную **ATTiny26** — она и стоит дороже и фиг где купишь у нас, да и что тогда делать с той прорвой **ATTiny2313** что уже закуплена? Пораскинул мозгами…

А почему бы не сварганить **АЦП** последовательного сравнения? Конечно, быстродействие и точность будет не фонтан, зато, не меняя тип МК и всего с двумя копеечными деталями дополнительного обвеса, я получу полноценный, хоть и тормозной, 8ми разрядный АЦП, вполне удовлетворяющий моим скромным запросам!

Как работает АЦП последовательного сравнения.
Что у нас есть в **ATTiny2313** аналогового? Правильно — **аналоговый компаратор** [1]. Теперь достаточно подать на его вход замеряемый сигнал и методично сравнивать с опорным напряжением, линейно изменяя величину опорного напряжения. На каком из опорных напряжений произойдет сработка компаратора, тому и примерно равен измеряемый сигнал +/_ шаг изменения опорного.

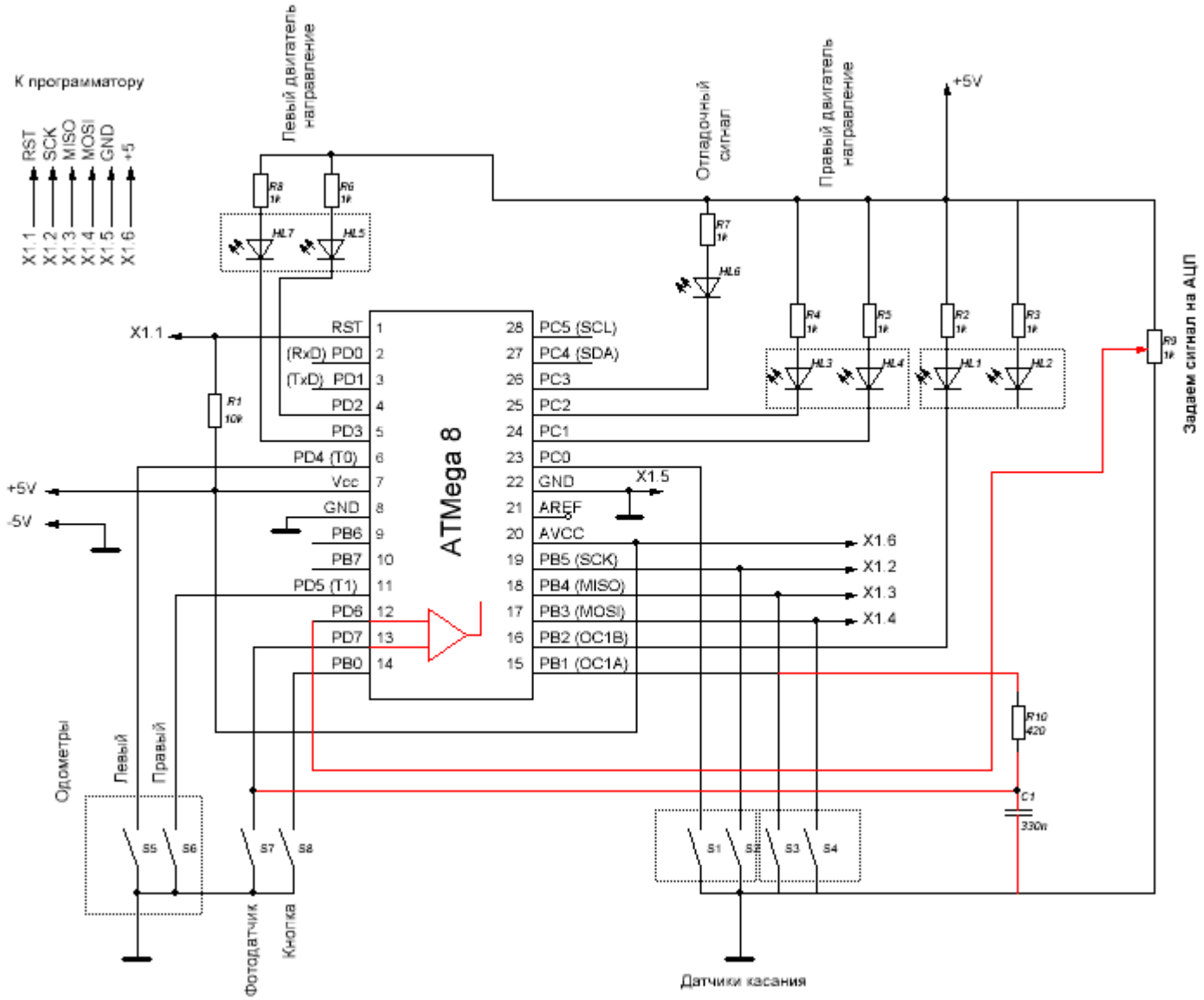


Осталось получить переменное опорное напряжение, а чем, из сугубо цифрового выхода контроллера, можно вытянуть аналоговый сигнал? ШИМом! Предварительно его проинтегрировав. Для интеграции используем простейший RC фильтр. Конденсатор у нас будет интегрировать заряд, а резистор не даст сдохнуть порту при зарядке кондера. Результатом прогона ШИМ'а через подобный фильтр станет достаточно стабильное постоянное напряжение.

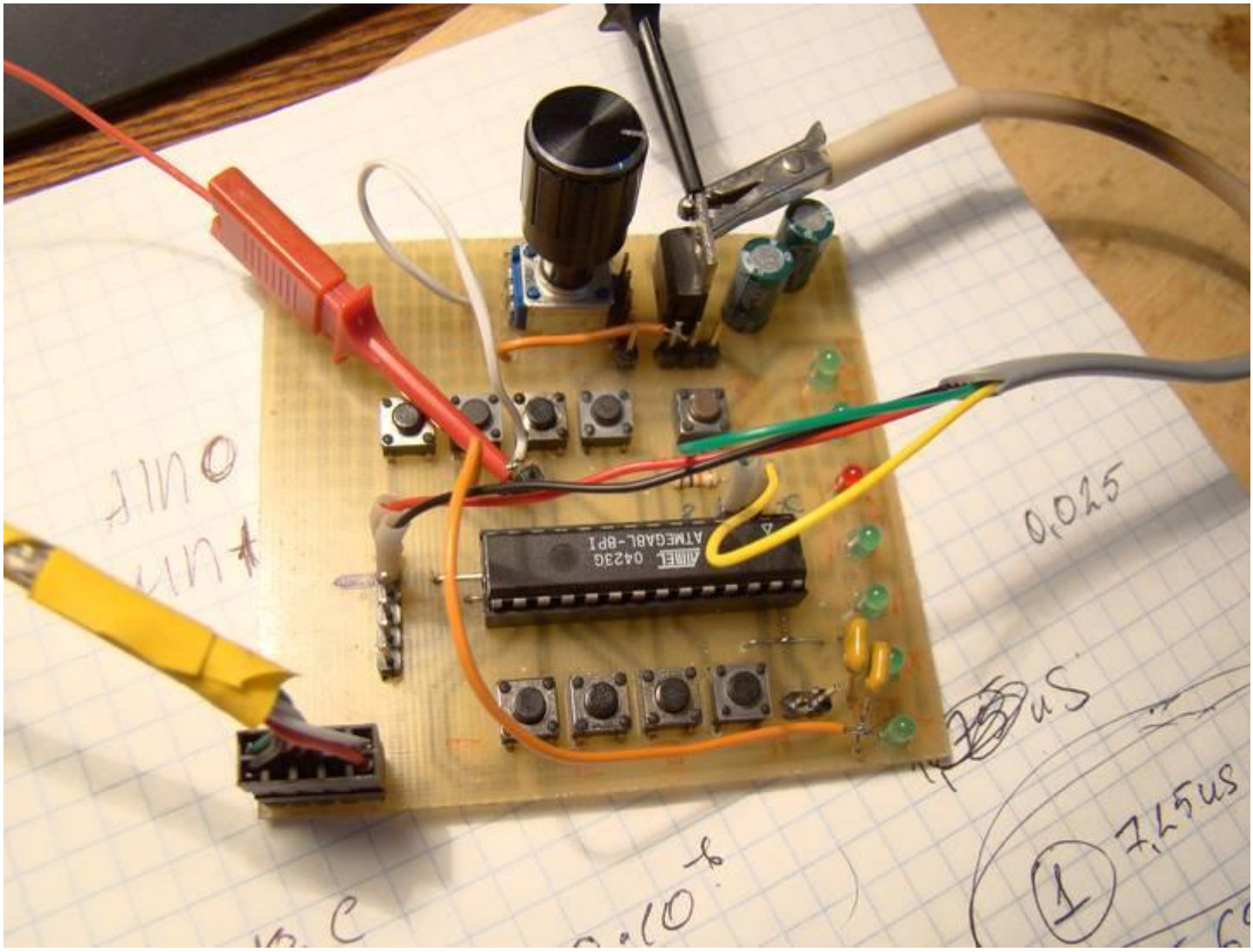
Осталось только прикинуть номиналы фильтра. Частота среза — частота, начиная с которой, фильтр начинает глушить переменную составляющую, у Г образного RC фильтра равна обратной величине из его постоянной

времени **w=1/RC**. Я воткнул кондер на **0.33E-6** Ф и резистор на **470** Ом, получилось что **w=6447 рад/с**. Поскольку **угловая частота** нам никуда не уперлась, то делим ее на 2pi = 6.28 получили около килогерца, 1026.6 Гц, если быть точным. Раз частота **ШИМ**а у нас запросто может быть порядка десятков килогерц, то на выходе будет гладенькая такая постоянка, с незначительными пульсациями.

Теперь заворачиваем эту ботву на вход компаратора, на второй пускаем наш измеряемый сигнал и начинаем развлекаться с кодом. Получилась вот такая схема, собранная на той же макетке [2], что и прошлый раз [3]. Тут, правда, не ATTiny2313, а Mega8 у которой АЦП есть, но мы пока забудем о его существовании. Красными линиями нарисован наш фильтр.



А это фотография платы. Резистор напаян снизу, на дорожки, а вот конденсаторы видно. Их два, так как я тут и на второй канал ШИМ повесил фильтр, правда так и не задействовал. Также видно, что белый провод от переменного резистора теперь идет на инверсный компаратора, а оранжевый идет с ноги конденсатора на прямой вход.



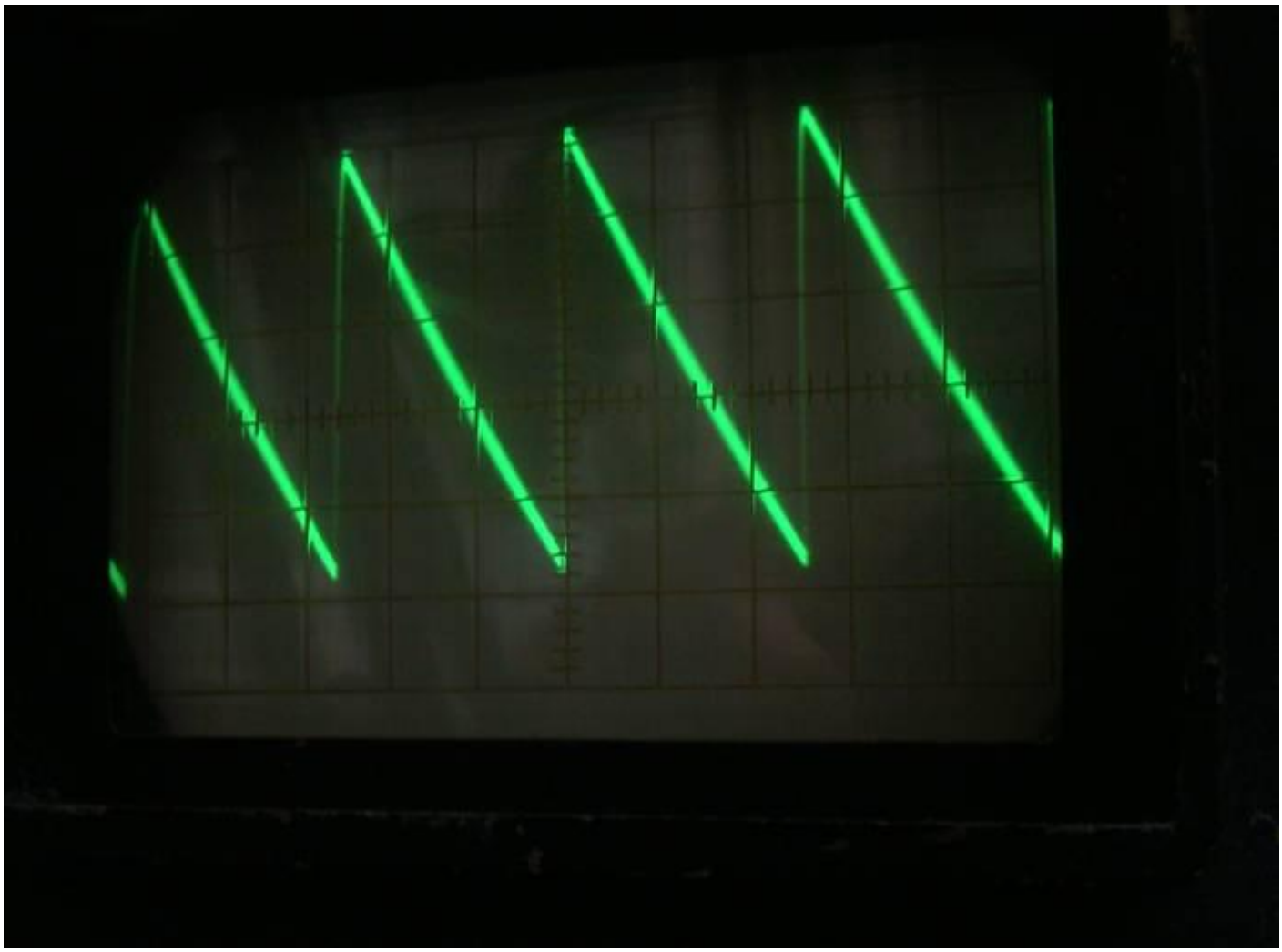
Код будет простецкий, чтобы не заморачиваться выложу **архив проекта** [4] и отдельные исходники в виде файлов:

- UARTundADC.asm [5] — Головной файл
- vectors.asm [6] — Таблица векторов прерываний
- init.asm [7] — Инициализация периферии
- macro.asm [8] и define.asm [9] — Макросы и макроопределения

Прокомментирую лишь главную функцию **Calc**.
При вызове процедуры **Calc** у нас первым делом:

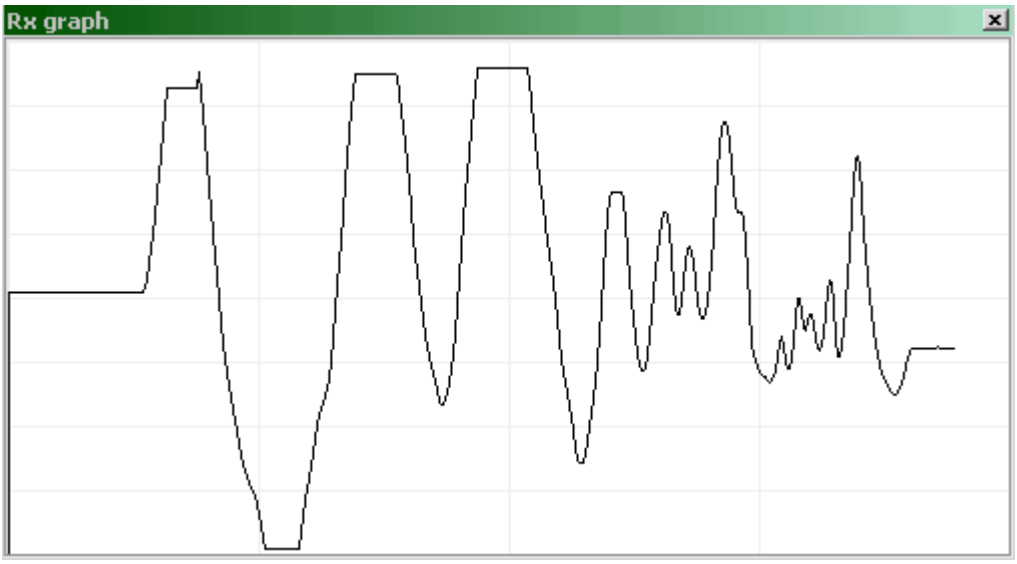
1. Конфигурируется аналоговый компаратор и, главное, активизируются его прерывания. Описание и настройка компаратора уже были описаны ранее [1]
2. Затем в сканирующий регистр (R21) закидывается начальное значение сканирования 255.
3. После чего это значение забрасывается в регистр сравнения ШИМ **OCR1AL**. ШИМ был заранее, в разделе **init.asm** сконфигурирован и запущен, так что сразу же на его выходе появляется сигнал скважностью (скважность это отношение длительности сигнала к периоду этого сигнала) 1 т.е., фактически, пока это просто единица.
4. Выжидаем в функции Delay некоторое время, чтобы закончился переходный процесс (конденсатор не может мгновенно изменить свое напряжение)
5. Уменьшем значение сканирующего регистра (что при загрузке в **OCR1AL** уменьшит скважность на 1/255), проверяем не стало ли оно нулю. Если нет, переходим на пункт 3.

Итогом станет последовательное уменьшение скважности сигнала с 1 до 0, с шагом в **1/255**, что будет преобразовано после фильтра в уменьшающееся напряжение. А, так как в главной процедуре у меня **Calc** вызывается циклически, то на входе компаратора будет пила.



А остальное сделает **компаратор**. Как только он сработает, а он сработает в любом случае — ведь какое нибудь да напряжение обязательно есть на входе. То его прерывание возьмет текущее значение регистра сравнения **OCR1AL**, ведь именно на нем у нас совпало напряжение, и выдаст нам в качестве результата замера.

По окончании скан цикла, выданное значение отправляется в регистр **UDR** и улетает в комп, рисуя в окне **Terminal'a** [10] график вращения ручки переменного резистора.



Как видно, вверху есть некоторый срез. Это связано с тем, что максимальное напряжение, которое может выдать нога МК, с учетом падений на всех резисторах, порядка 4.7 вольта, а с задающего потенциоматера я могу и все 5 выкрутить. Ну еще и верхушки заваливаются чуток. Если понизить частоту, то диапазон несколько расширится.

Вот так, применив немного смекалки, а также две дополнительные детали общей суммой в один рубль и десяток строк кода, я сэкономил кучу бабла =)

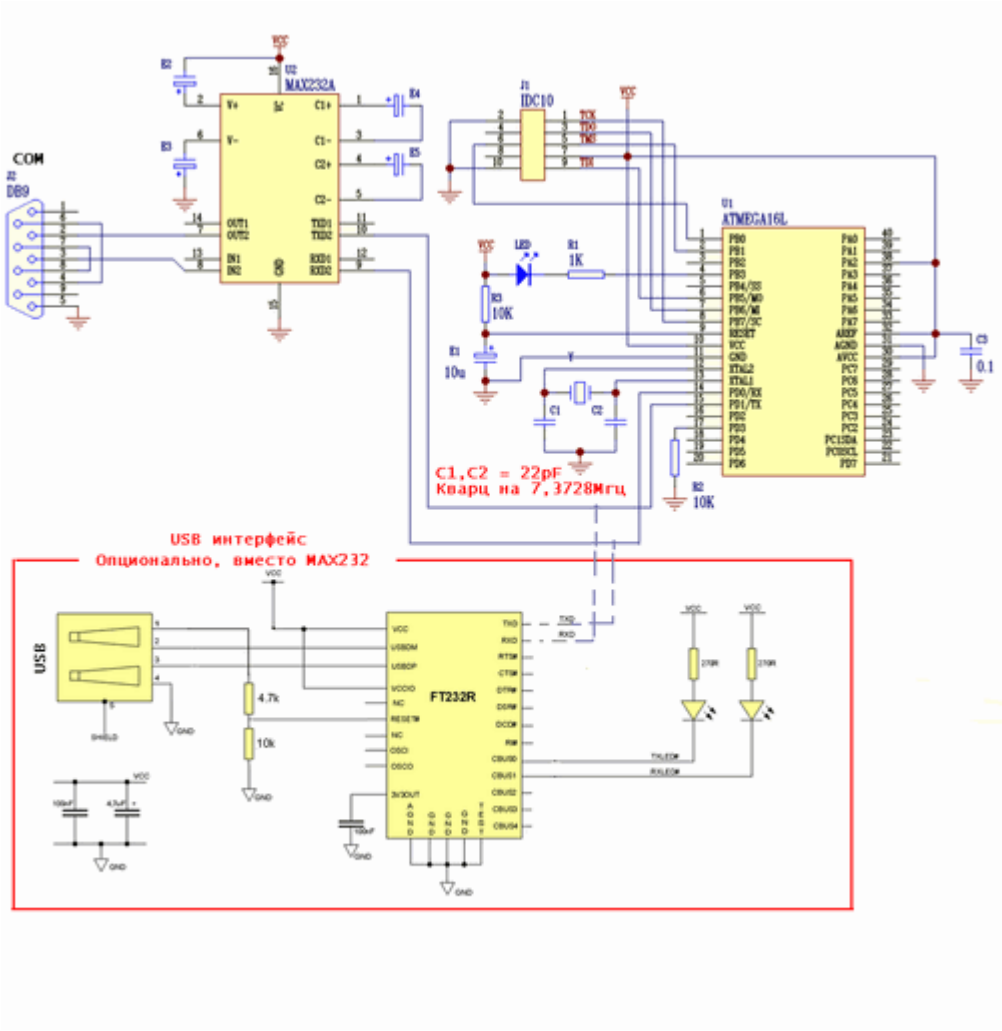
Внутрисхемная отладка AVR через JTAG ICE

Одним из самых мощных отладочных инструментов в микроконтроллерной среде является **JTAG — внутрисхемный отладчик**.

Суть тут в чем — через JTAG адаптер микроконтроллер подключается напрямую к среде программирвания и отладки, например к **AVR Studio**. После чего микроконтроллер полностью подчиняется студии и дальше шагу не может ступить без ее разрешения. Доступна становится пошаговое выполнение кода, просмотр/изменение всех регистров, работа со всей периферией и все это в реальном микроконтроллере, а не в программной эмуляции. Можно ставить точки останова (breakpoints) на разные события — для ICE1 три штуки. В общем, сказка, а не инструмент.

К сожалению в AVR микроконтроллерах JTAG доступен далеко не везде, как правило контроллеры с числом ног меньше 40 и обьемом памяти меньше 16КБ такого удовольствия лишены (там, правда, часто бывает debugWire, но на коленке сделать его адаптер еще никому не удалось, а фирменный JTAG ICEII или AVRDragon стоят довольно больших денег). А поскольку у меня в ходу в основном Tiny2313, Mega8, Mega8535 и прочая мелочевка, то мне JTAG что собаке пятая нога — не поддерживается он в этих МК.

Тем не менее, поддашвись многочисленным просьбам, я сварганил этот агрегат и сейчас покажу вам как им пользоваться.
Велосипед изобретать я не стал и взял широко известный проект от scienceprog.com [1]



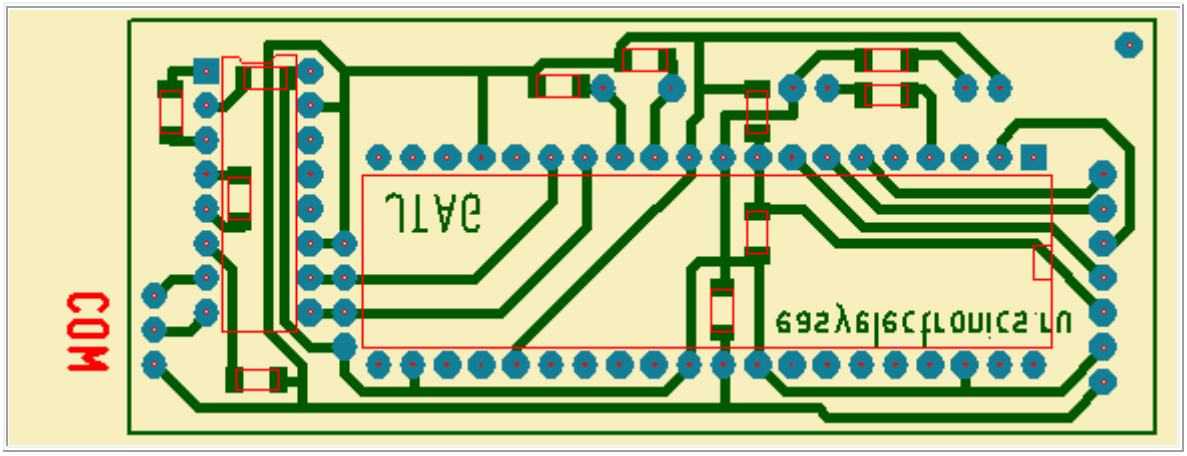
Увеличить схему [2]

Схема несложная, требует **ATMega16** и чуть чуть обвяза. В качестве интерфейса можно поставить **MAX232** и воткнуть все это дело в COM порт, а можно сделать на **FT232RL** и тогда интерфейсом будет **USB**.

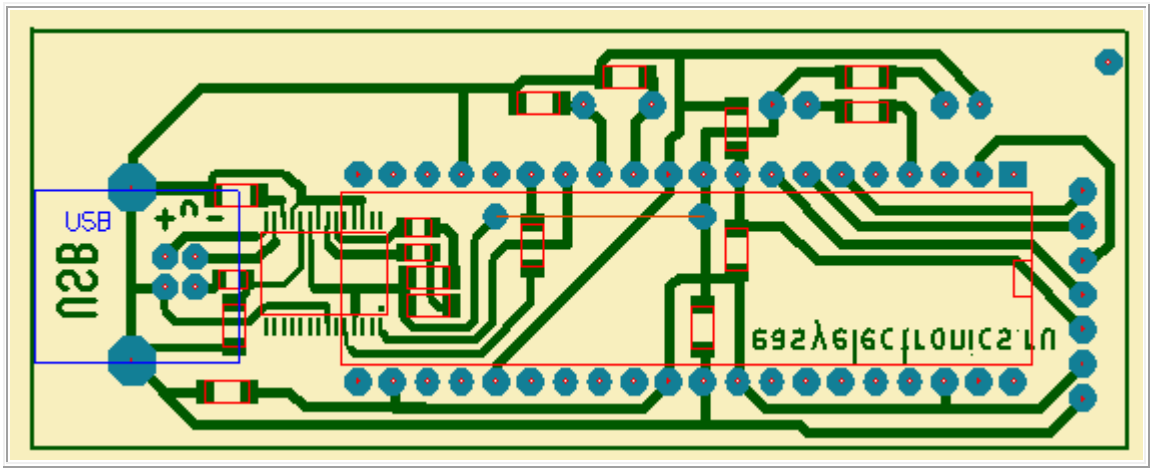
Только если будете делать на **FT232** то рекомендую питание брать не от **USB**, а от целевого устройства. Дело в том, что если подключишь **JTAG** на **FT232** с запиткой от **USB** к незапитанной целевой схеме (например по ошибке или недосмотру) то целевая схема запитается через защитные диоды паразитным питанием [3], т.к. на выводах **FT232** будет высокий уровень. Что черевато тем, что может сдохнуть либо **JTAG** контроллер, либо контроллер целевой платы.

Поэтому пусть лучше **JTAG** адаптер питается от целевой платы, а не наоборот.

Я же не стал мудрить и сделал на MAX232 — с USB у меня напряг вечный, а COM порты свободны. Тем более это дешевле.

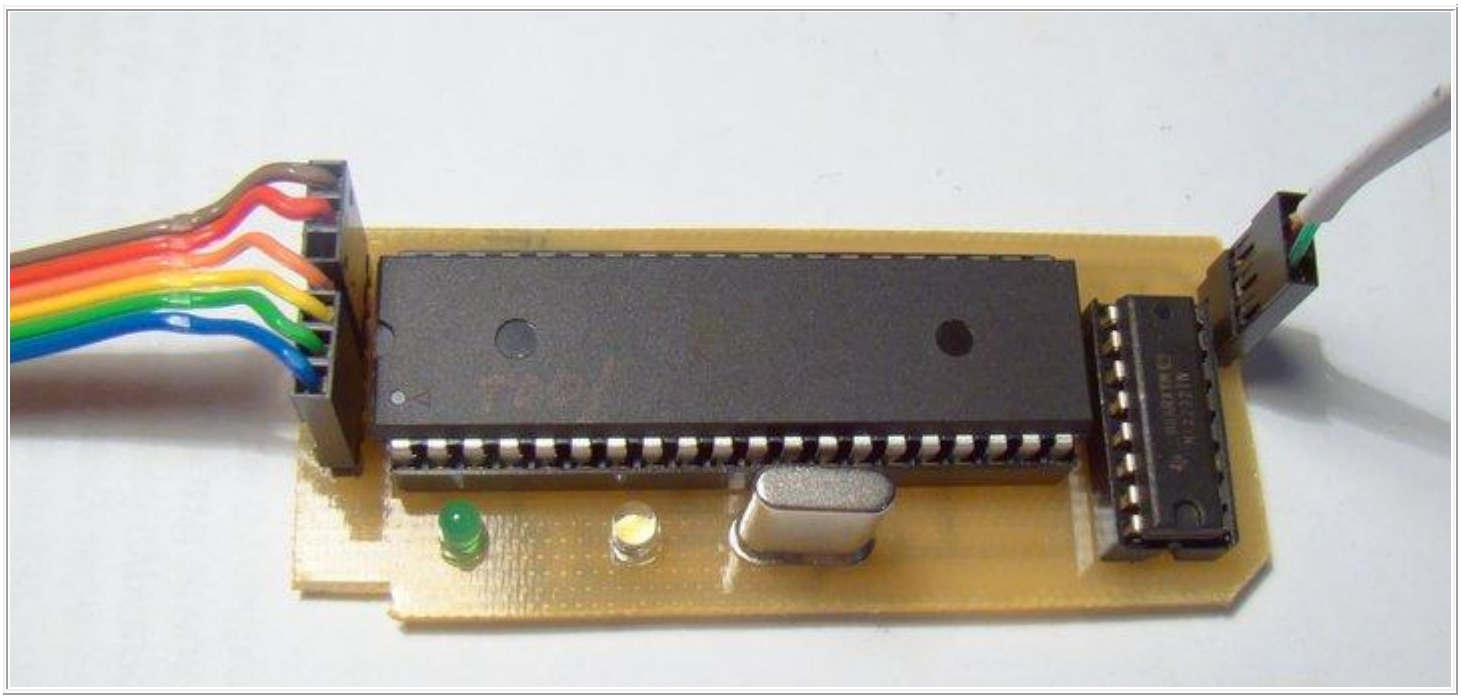


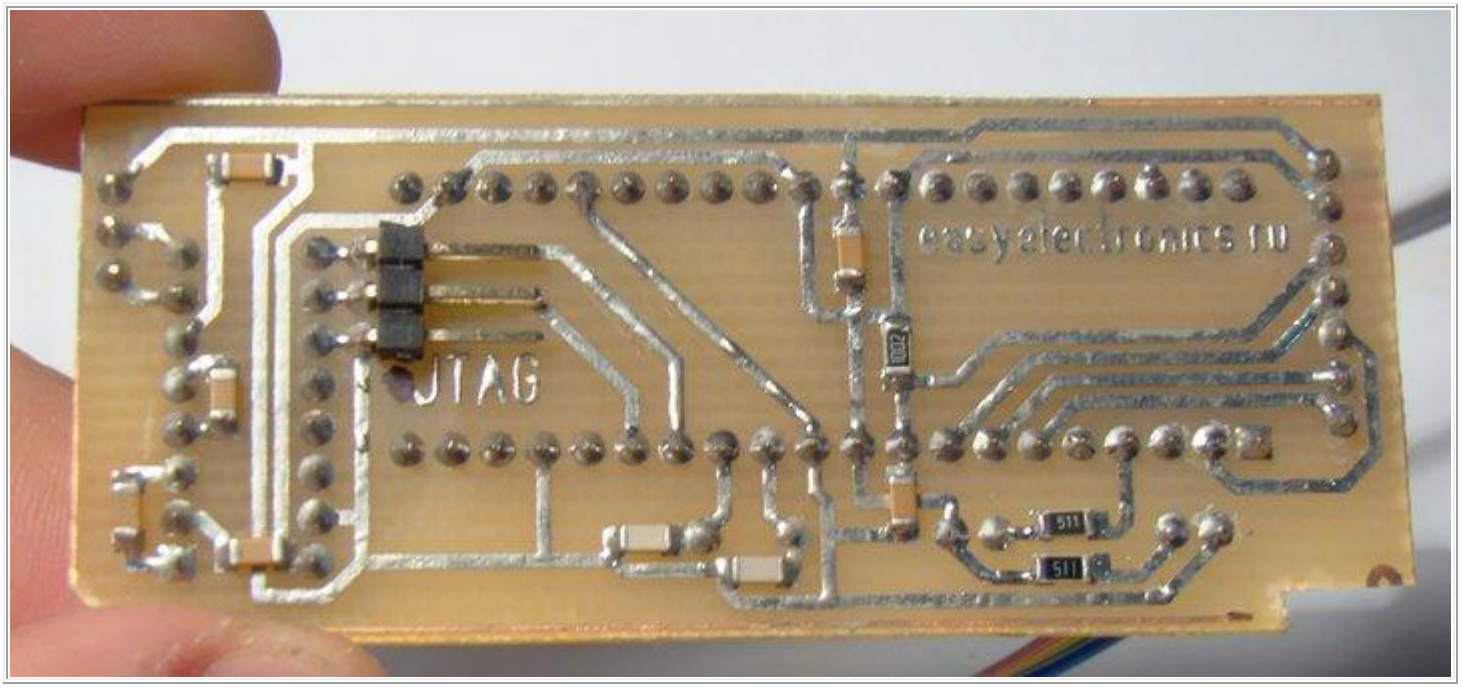
Однако я не обломался развести плату и под FT232RL суть та же.



В остальном же, как видите, разводка почти один в один, а **USB** часть взял и своего прошлого [поста про **FT232**](#) [4]. Кстати, разводя вот такие вот маленькие кусочки схем [вроде блока с **FT232**](#) [4] я их сохраняю в макросы и потом, когда надо, собираю из них печатную плату как из конструктора. Очень удобно! Рекомендую!

Вытравили плату, запаяли компоненты. Получилась вот такая вот платка:





Теперь надо прошить контроллер. Вообще, по науке, сначала прошивается **bootloader**, потом из **AVR Studio** делается обновление прошивки **JTAG ICE** до последней версии. Но я нашел путь проще, на сайте pol-sem.narod.ru [5] был обнаружен уже готовый HEX файл который просто надо залить в МК и все.

Поэтому цепляем к нашему адаптеру программатор, благо все штыри нужные (**MISO, MOSI, GND, RST, SCK, Vcc**) есть. А то что они не в нужном порядке стандартного **AVR ISP** разъема, так это не беда — я временно распотрошил колодку своего программатора и одел отдельные пины как надо. Однократная же процедура, чего мучаться разводить еще и ISP разъем.

Заливаем прошивку.
Выставляем Fuse биты.

Тут надо быть внимательными, так как существует несколько нотаций FUSE — прямая (по даташиту, где 0 = ON, 1=OFF) и инверсная (1 = ON, 0 = OFF). В прямой нотации работает UNIPROF, в инверсной нотации работает PonyProg и USBASP _AVRDUDE_PROG.
Определить в какой нотации работает твой программатор очень просто. Дjстаточно подключиться к своему МК и нажать кнопку **чтения** Fuse битов и посмотреть на бит **SPIEN** если галка **стоит** — нотация **инверсная**. Потому как по дефолту SPIEN включен всегда (без него невозможно прошить МК через ISP внутрисхемно).

Прошиваются Fuse следующим образом:

Бит	Прямая нотация (UniProf, Даташит)	Инверсная нотация (PonyProg, AVR DUDE GUI)
OCDEN	[ ]	[v]
JTAGEN	[ ]	[v]
SPIEN	[ ]	[v]
CKOPT	[v]	[ ]
EESAVE	[v]	[ ]
BOOTSZ1	[ ]	[v]
BOOTSZ0	[ ]	[v]
BOOTRST	[v]	[ ]
BODLEVEL	[v]	[ ]
BODEN	[v]	[ ]

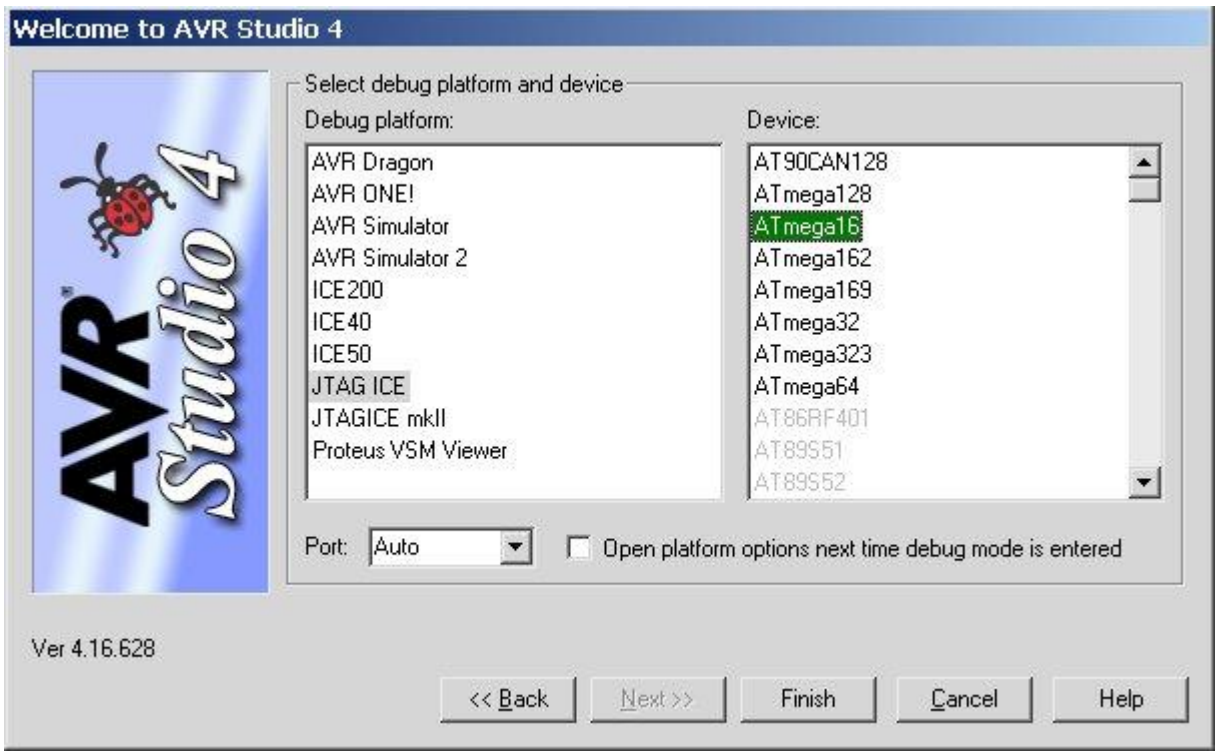
SUT1	[ ]	[v]
SUT0	[ ]	[v]
CKSEL3	[v]	[ ]
CKSEL2	[v]	[ ]
CKSEL1	[v]	[ ]
CKSEL0	[v]	[ ]

Если прошивать голый бутлоадер, то надо включить бит BOOTRST и подключившись через студию сделать обновление прошивки JTAG, залив через AVRProg файл upgrade.ebn (лежит он где то в каталоге AVR Studio). А после прошивки выключить BOOTRST.

Все, девайс готов к работе. Теперь осталось его только испытать в деле.

Работа с JTAG AVR ICE
Особо заморачиваться я не буду, так что покажу на простом примере мигания светодиодов.
Запускаем студию, создаем новый проект.
Выбираем язык программирования, пусть это будет Assembler.
Задаем имя проекта.

А далее, в разделе Debug Platform выбираем не AVR Simulator как раньше, а JTAG ICE.



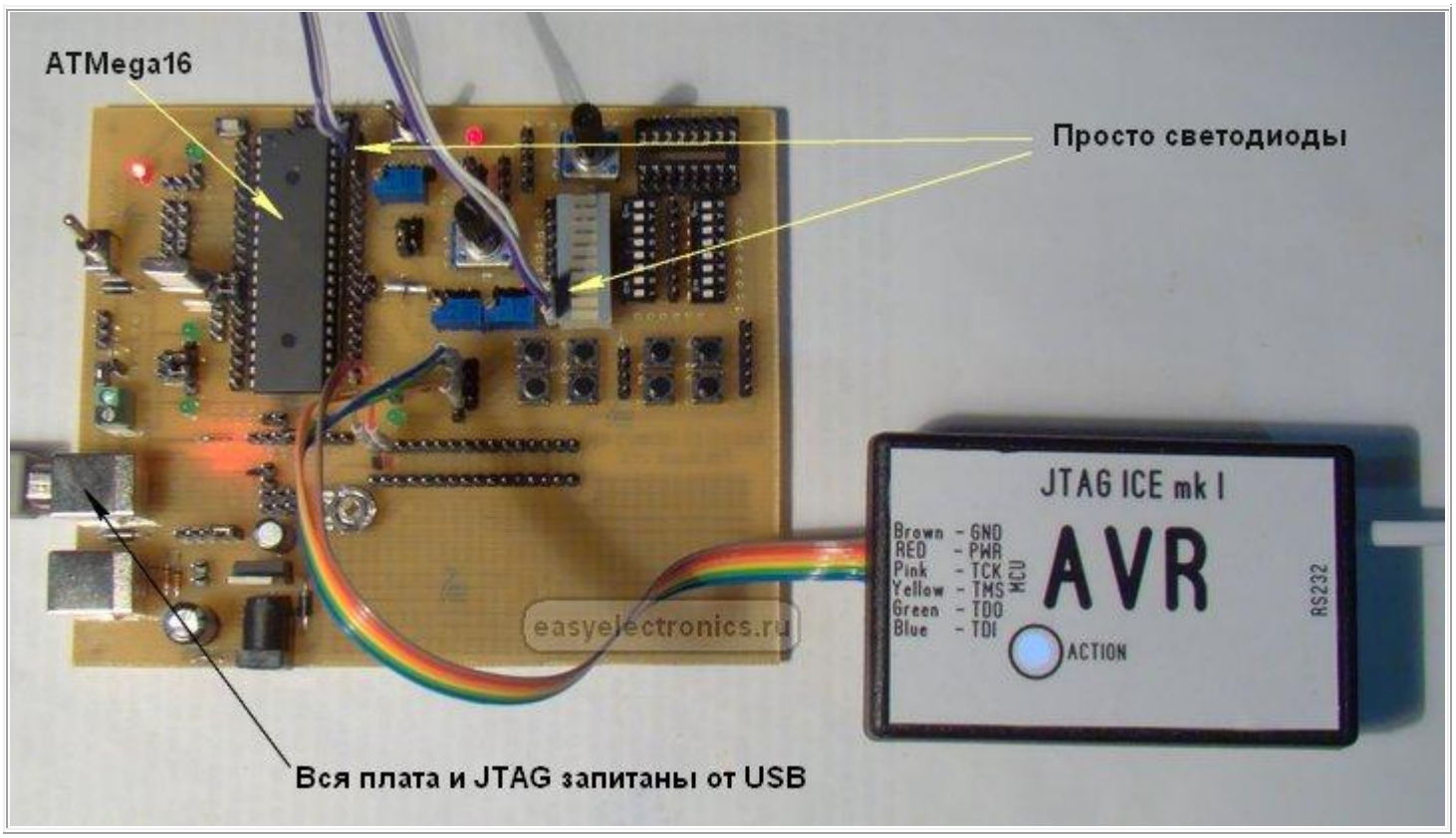
В правом окне выбираем отлаживаемый кристалл, (у меня это Mega16) и жмем финиш. Все, дальше как обычно, вбиваем текст программы. Я не стал мудрить и по быстрому настрогал следующее:

```
1          .include "m16def.inc"  ; Используем ATMega16
2
3
4          LDI     R16,0xFF              ; Порт A на выход.
5          OUT     DDRA,R16
6
7  Main:   SEI                   ; Разрешаем прерывания.
8
9
```

```
10          SBI     PORTA,0         ; Зажгли диод 0
11          SBI     PORTA,1         ; Зажгли диод 1
12          SBI     PORTA,2         ; Зажгли диод 2
13          NOP
14          CBI     PORTA,0         ; Погасили диод 0
15          CBI     PORTA,1         ; Погасили диод 1
16          CBI     PORTA,2         ; Погасили диод 2
17          NOP
18
19          RJMP    Main            ; Зациклились
```

Вот так вот просто. Если ее скомпилить, прошить и запустить, то диоды будут моргать с бешеной частотой, так как никаких задержек не предусмотрено. А что же будет из под **JTAG**?

Беру свою новоиспеченную отладочную плату на **Mega16**, подключаю к порту А три светодиода.
Подключаю к плате **JTAG** адаптер — четыре провода интерфейса (**TDO,TDI,TMS,TCK**) и два силовых Vcc на плюс и GND на землю — **JTAG** адптер запитан от целевой платы и все готово к работе.

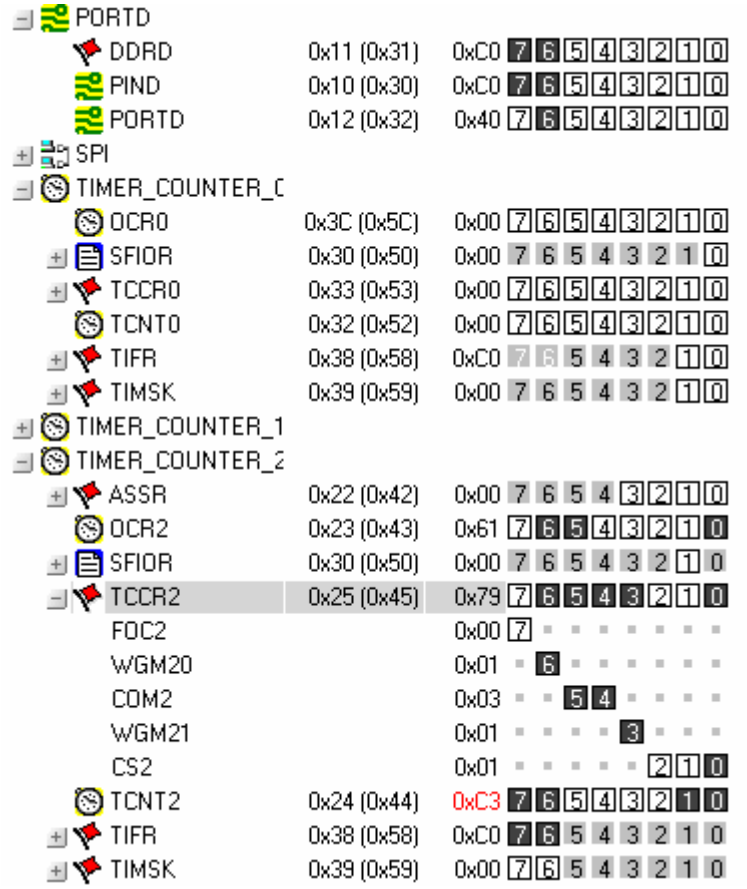


Жму в студии компиляцию и запуск (Ctrl+F7). Проект по быстрому компилится, тут же через **JTAG** заливается в МК (JTAG может заменить программатор) адаптер при этом весело перемигивается светодиодом. Программа встает на выполнение, бодро показывая стрелочкой начало программы.

```
                .include "m16def.inc"    ; Используем ATMega8

                LDI     R16,0xFF
                OUT     DDRA,R16

Main:           SEI                      ; Разрешаем прерывания.

                SBI     PORTA,0
                SBI     PORTA,1
                SBI     PORTA,2
                NOP
                CBI     PORTA,0
                CBI     PORTA,1
                CBI     PORTA,2
                NOP

                RJMP    Main
```

Можно трассировать! Тыкаю по F11 — прога исполняется по одной команде, показывая стрелочкой где я нахожусь в данный момент. После каждого выполнения команды SBI — у меня на плате зажигается соответствующий светодиод. Круто, блин! Как в каком-нибудь **Proteus** [6] только без глюков и все вживую! Пробежался дальше по тексту — после CBI диоды погасли, как и положено. Вот как это выглядит вживую:

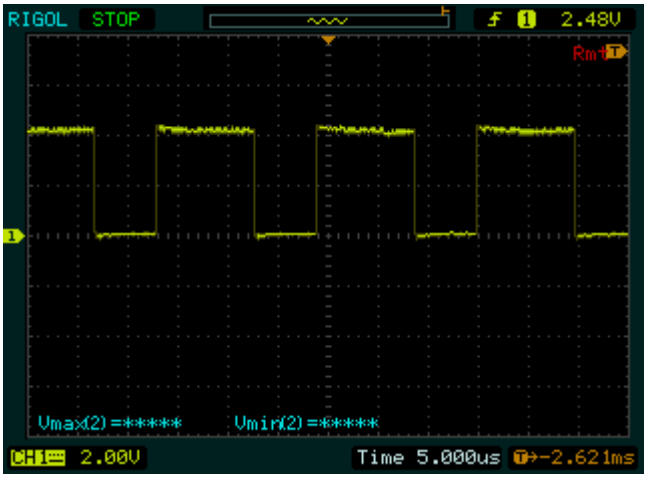
Дальше решил поразвлекаться. Не меняя программу, не перекомпиливая, не выходя из режима отладки. Открываю в **AVR Studio** вкладку **I/O View**
Протыкиваю там галочки:



- DDRD.7=1 — вывод PD7 на выход
- Затем лезу в раздел Timer_Counter_2 и там прямыми тычками по битам выставляю:
- COM2 = 11 — инверсный режим работы вывода OC2 в режиме FastPWM
- WGM20 = 1, WGM21 = 1 — Режим таймера 2 устанавливаем в FastPWM

- От балды натыкиваю число в OCR2 — от него зависит коэффициент заполнения
- Выставляю биты CS2 = 001 — запуск таймера.

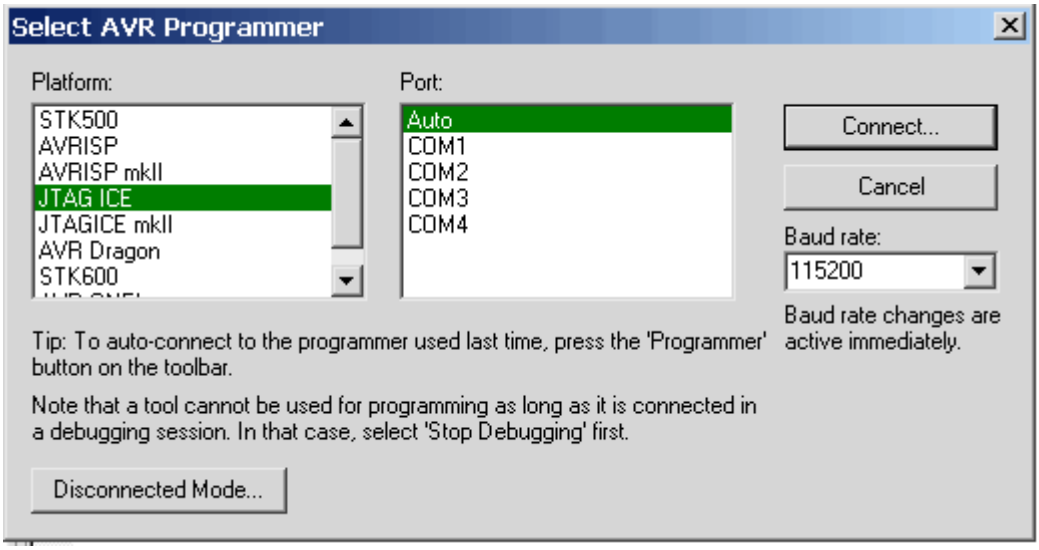
Снимаю прогу с паузы (F5 — Run). Тычу осциллографом в ногу PD7 (OC2)



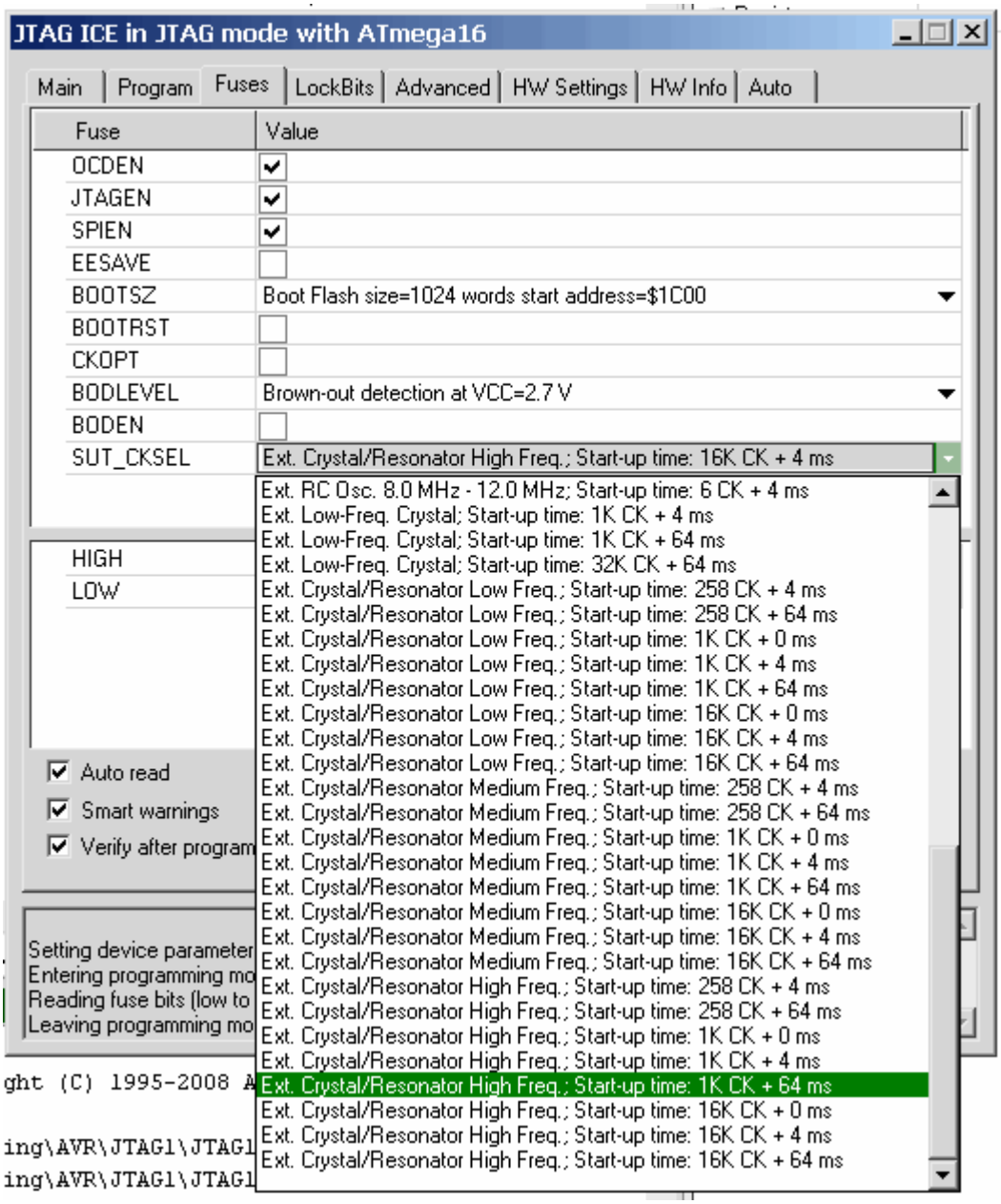
ЫЫЫ!!! ШИМ!!! Ставлю прогу на паузу, меняю биты в OCR2 запускаю снова — коэффициент заполнения изменился. Хы. Ручное управление :)

Так что с JTAGом если хочешь получить по быстрому какую нибудь фиговину вроде генератора даже не надо прогу писать — взял и включил вручную что тебе нужно. Богат AVR периферией :)

Прошивка микроконтроллера через JTAG
JTAG ICE можно также использовать для прошивки микроконтроллера и установки FUSE битов. Для этого надо запустить **AVR PROG** (Tools — Program AVR — Connect…) и выбрать там JTAG ICE ну и COM порт, хотя обычно канает Auto.



Жмем **Connect** и мы в эфире, главное не забыть выйти из режима откладки в студии. AVRPROG сам определяет тип контроллера, можно выбрать файл с флешем и/или епромом и прошить их. Отдельного разговора заслуживает вкладка Fuses.
В отличии от большинства программаторов, тут уже не придется париться по поводу битов SUT:CKSEL — все выбирается из понятного списка, написанного человеческим языком:



Наигрался, зафотографировал для статьи, упаковал в красивый корпус. Даже не поленился и обложечку сфигачил. Очень уж мне игрушка понравилась.

Файлы к статье:

- Печатная плата в формате Sprint Layout [7]
- Прошивка (целиком) [8]

Вот так и развращаютя эмбеддеры. Сначала к JTAG пристрастятся, потом ради одного лишь JTAG преeходят на более мощный кристалл там, где хватит и Tiny, а дальше Си, Си++, потом .NET какой нибудь на виртуальной машине… И вот уже операционная система весом в несколько гигабайт и требующая гигазы ОЗУ ни у кого не вызывает шока и ужаса. А ведь это страшно, господа! Прогресс, мать его. Не развращайтесь благами цивилизации, будте аскетичны и разумны. И не забывайте об оптимизации как программной, так и аппаратной.

AVR. Учебный Курс. Программирование на Си. Часть 1

Я не раз и не два говорил, что изучение МК надо начинать с ассемблера. Этому был посвящен целый курс на сайте (правда он не очень последовательный, но постепенно я его причесываю до адекватного вида) . Да, это сложно, результат будет не в первый день, но зато ты научишься понимать что происходит у тебя в контроллере. Будешь знать как это работает, а не по обезьяньий копировать чужие исходники и пытаться понять почему оно вдруг перестало работать. Кроме того, Си намного проще натворить быдлокода, который вылезет вилами в самый неподходящий момент.

К сожалению все хотят результат немедленно. Поэтому я решил пойти с другой стороны — сделать обучалку по Си, но с показом его нижнего белья. Хороший программист-эмбеддер всегда крепко держит свою железку за шкварник, не давая ей ни шагу ступить без разрешения. Так что будет вначале Си код, потом то что родил компилятор и как все это работает на самом деле :)

С другой стороны у Си сильная сторона это переносимость кода. Если, конечно, писать все правильно. Разделяя алгоритмы работы и их железные реализации в разные части проекта. Тогда для переноса алгоритма в другой МК достаточно будет переписать только интерфейсный слой, где прописано все обращение к железу, а весь рабочий код оставить как есть. И, конечно же, читаемость. Сишный исходник проще понять с первого взгляда (хотя.. мне, например, уже пофигу на что фтыкать — хоть си, хоть асм :)), но, опять же, если правильно все написать. Этим моментам я тоже буду уделять внимание.

В качестве подопытной железки на которой будет ставиться львинная доля всех примеров будет моя отладочная плата PinBoard [1].

Дальше все будет разжевано буквально по шагам для старта с полного нуля.

Первая программа на Си для AVR

Выбор компилятора и установка среды
Для AVR существует множество разных компиляторов Си:
В первую очередь это **IAR AVR C** — почти однозначно признается лучшим компилятором для AVR, т.к. сам контроллер создавался тесном сотрудничистве Atmel и спецов из IAR. Но за все приходится платить. И этот компилятор мало того, что является дорогущим коммерческим софтом, так еще обладает такой прорвой настроек, что просто взять и скомпилить в нем это надо постраться. У меня с ним правда не срослось дружбы, проект загнивал на странных ошибках на этапе линковки (позже выяснил, что это был кривой кряк).

Вторым идет **WinAVR GCC** — мощный оптимизирующий компилятор. Полный опенсорц, кроссплатформенный, в общем, все радости жизни. Еще он отлично интегрируется в AVR Studio позволяя вести отладку прямо там, что адски удобно. В общем, я выбрал его.

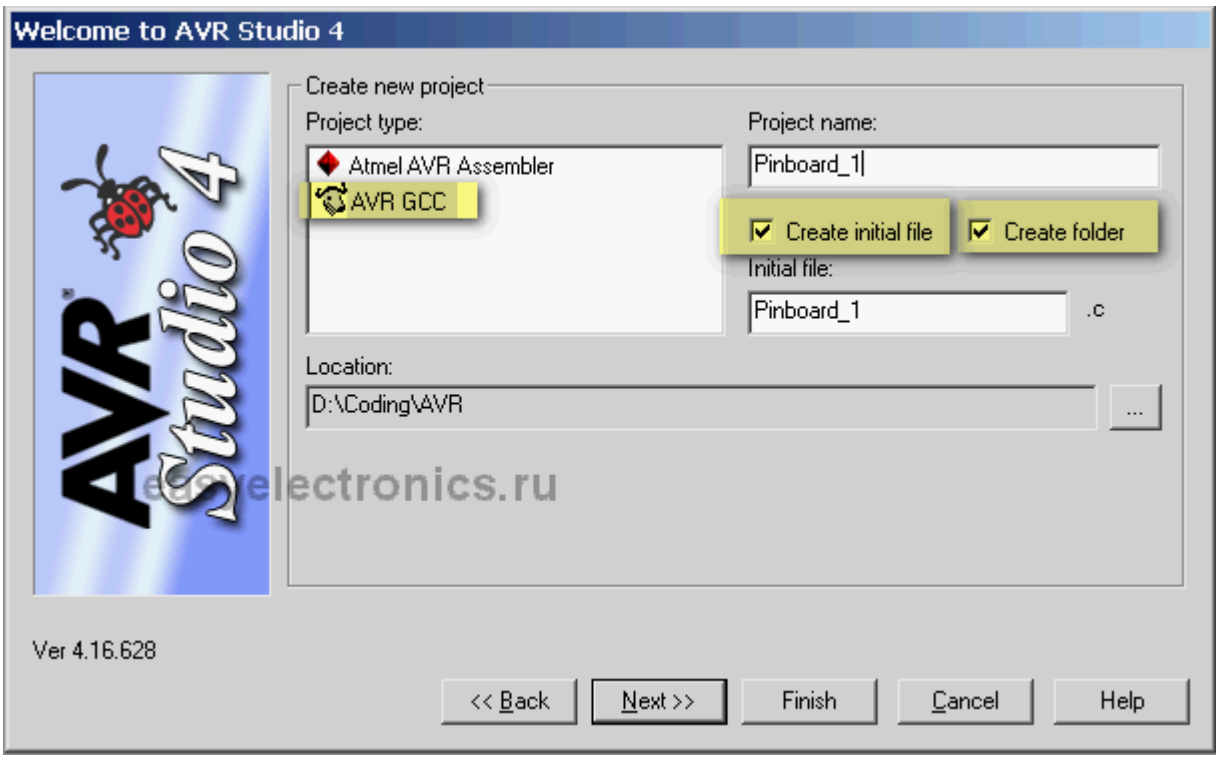
Также есть **CodeVision AVR C** — очень популярный компилятор. Стал популярен в связи со своей простотой. Рабочую программу в нем получить можно уже через несколько минут — мастер стартового кода этом сильно способствует, штампуя стандартыне инициализации всяких уартов. Честно говоря, я как то с подозрением к нему отношусь — как то раз приходилось дизасмить прогу написаную этим компилером, каша какая то а не код получалась. Жуткое количество ненужных телодвижений и операций, что выливалось в неслабый обьем кода и медленное быстродействие. Впрочем, возможно тут была ошибка в ДНК писавшего исходную прошивку. Плюс он хочет денег. Не так много как IAR, но ощутимо. А в деморежиме дает писать не более чем 2кб кода.
Кряк конечно есть, но если уж воровать, так миллион, в смысле IAR :)

Еще есть **Image Craft AVR C** и **MicroC** от микроэлектроники. Ни тем ни другим пользоваться не приходилось, но вот **SWG** очень уж нахваливает **MicroPascal**, мол жутко удобная среда программирования и библиотеки. Думаю MicroC не хуже будет, но тоже платный.

Как я уже сказал, я выбра **WinAVR** по трем причинам: халявный, интегрируется в AVR Studio и под него написана просто прорва готового кода на все случаи жизни.

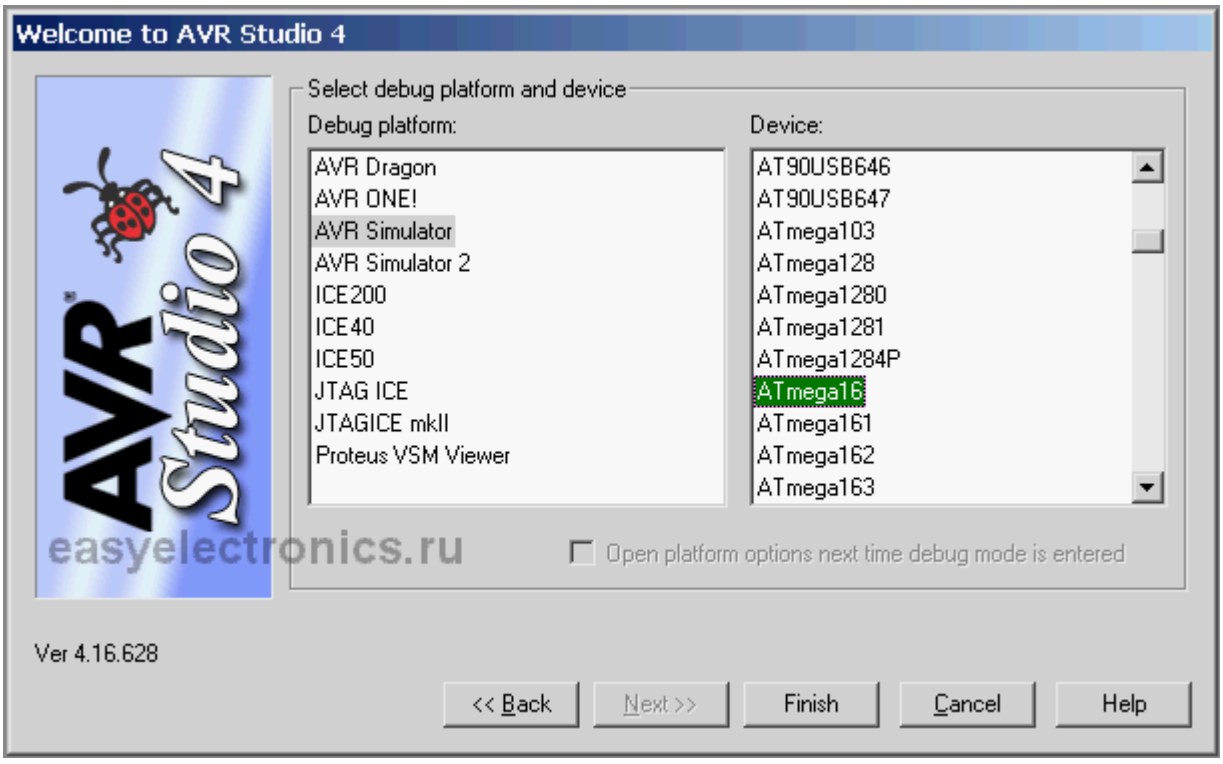
Так что качай себе инсталяху WinAVR с официального источника [2] и AVR Studio. Далее вначале ставится студия, потом, сверху, накатывается WinAVR и цепляется к студии в виде плагина. Настоятельно рекомендую ставить WinAVR по короткому пути, что то вроде C:\WinAVR тем самым ты избежишь кучи проблем с путями.

Создание проекта
Итак, студия поставлена, Си прикручен, пора бы и попробовать что нибудь запрограммировать. Начнем с простого, самого простого. Запускай студию, выбирай там новый проект, в качестве компилятора AVR GCC и вписывай название проекта.



Также не забудь поставить галочу Create Folder, чтобы у тебя все сложилось в одной директории. Ну и укажи место Location, где будет лежать проект. Указывай по короткому пути, что то вроде C:\AVR\ Как показывает практика, чем короче путь тем лучше — меньше проблем при компиляции и линковке проектов.

Проц у меня в Pinboard по дефолту **ATmega16**, поэтому выбираю его. Те же у кого в PinBoard стоит Mega32 (по спец заказу ставил некторым :)) выбирают, соответственно ее.

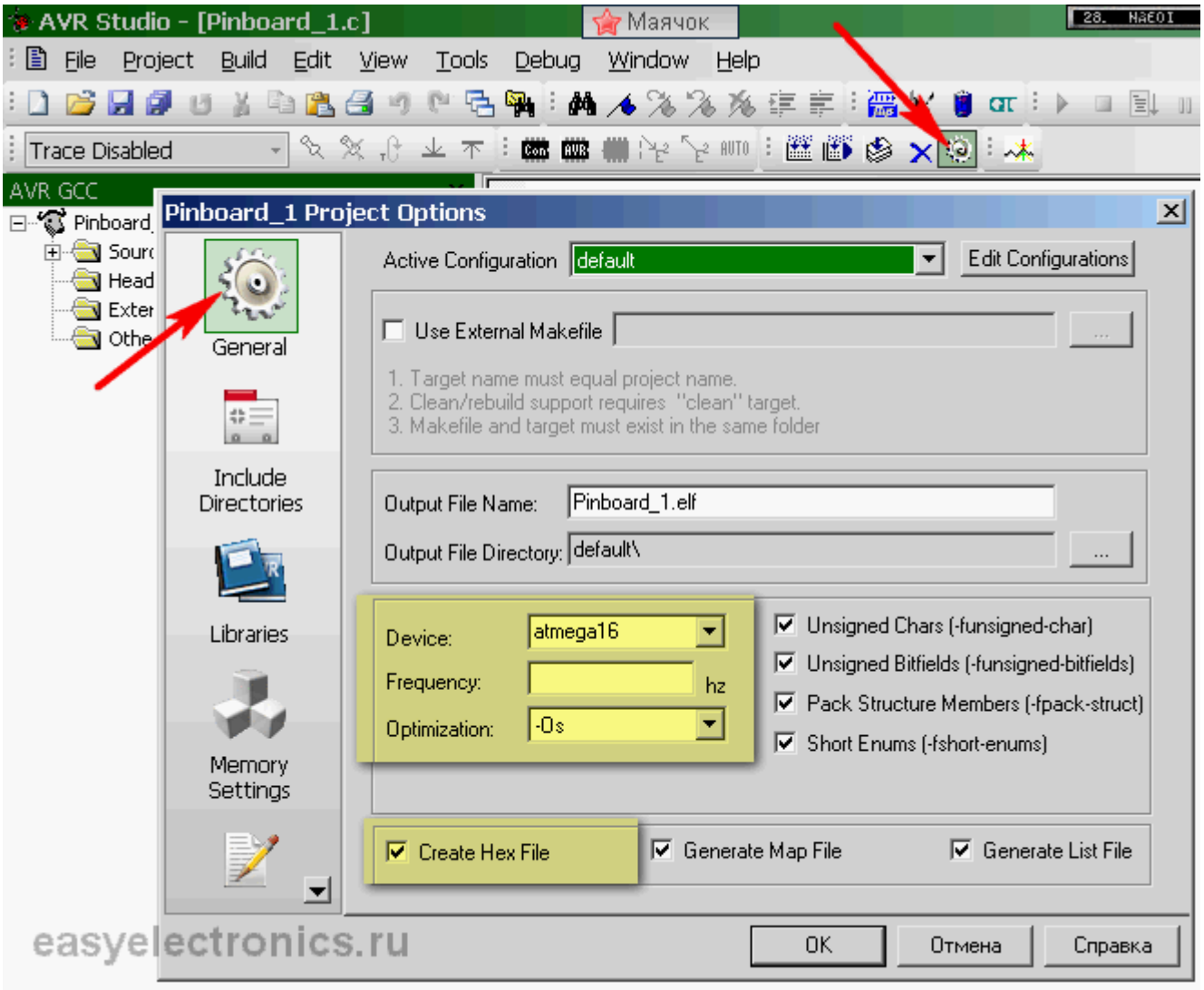


Открывается рабочее поле с пустым *.c файлом.

Теперь не помешает настроить отображение путей в закладках студии. Для этого слазь по адресу:
Меню Tools — Options — General — FileTabs и выбираем в выпадающем списке «Filename Only». Иначе работать будет невозможно — на вкладке будет полный путь файла и на экране будет не более двух трех вкладок.

Настройка проекта
Вообще, классическим считается создание make файла в котором бы были описаны все зависимости. И это, наверное, правильно. Но мне, выросшему на полностью интегрированных IDE вроде **uVision** или **AVR Studio** этот подход является глубоко чуждым. Поэтому буду делать по своему, все средствами студии.

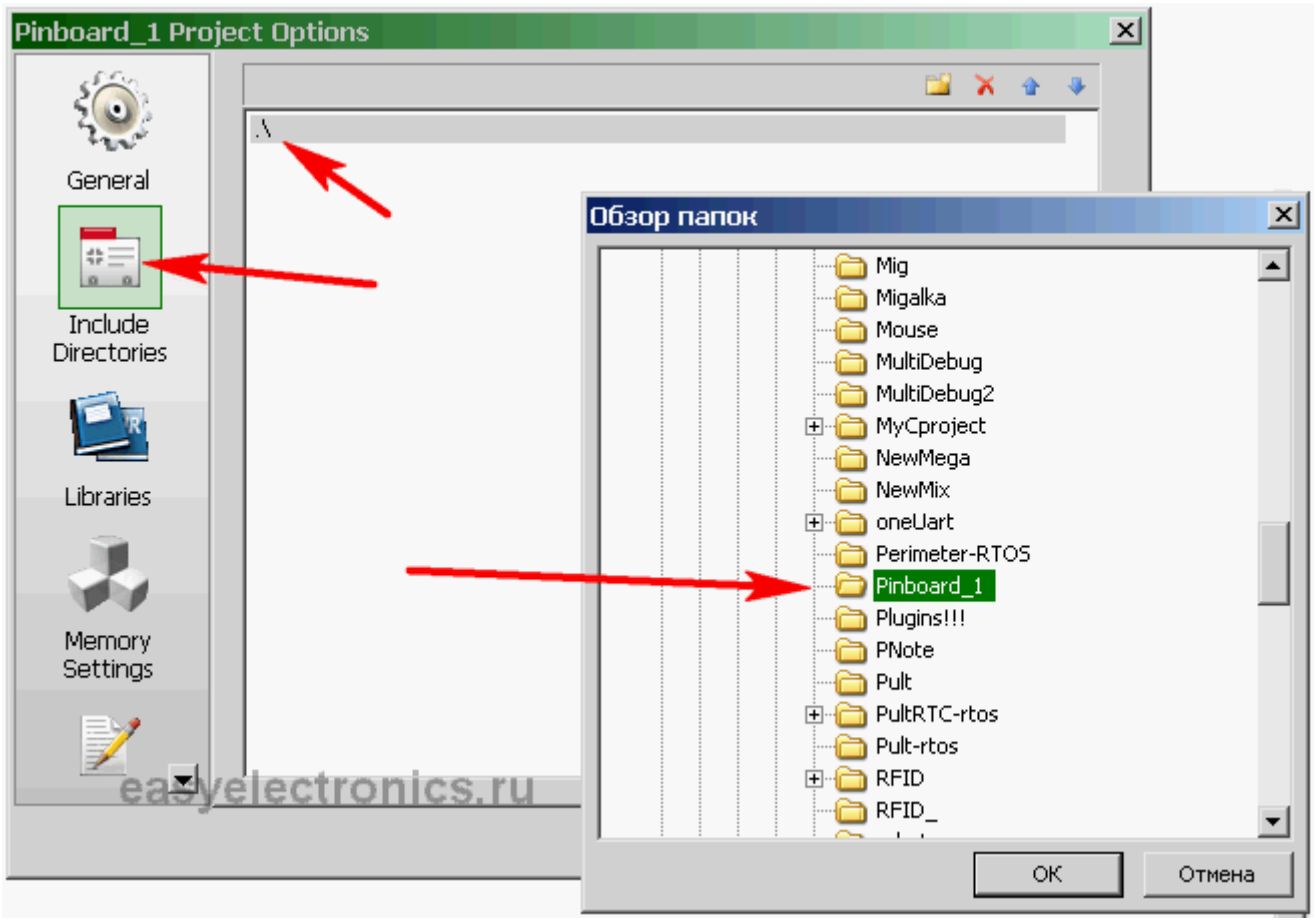
Тыкай в кнопку с шестеренкой.



Это настройки твоего проекта, а точнее настройки автоматической генерации make файла. На первой странице надо всего лишь вписать частоту на которой будет работать твой МК. Это зависит от фьюз битов, так что считаем что частота у нас 8000000Гц.
Также обрати внимание на строку оптимизации. Сейчас там стоит -Os это оптимизация по размеру. Пока оставь как есть, потом можешь попробовать поиграться с этим параметром. -O0 это отстутсвие оптимизации вообще.

Следующим шагом будет настройка путей. Первым делом добавь туда директорию твоего проекта — будешь туда подкладывать сторонние библиотеки. В списке появится путь «.\»

Make файл сгенерирован, его ты можешь поглядеть в папке default в своем проекте, просто пробегись глазами, посмотри что там есть.

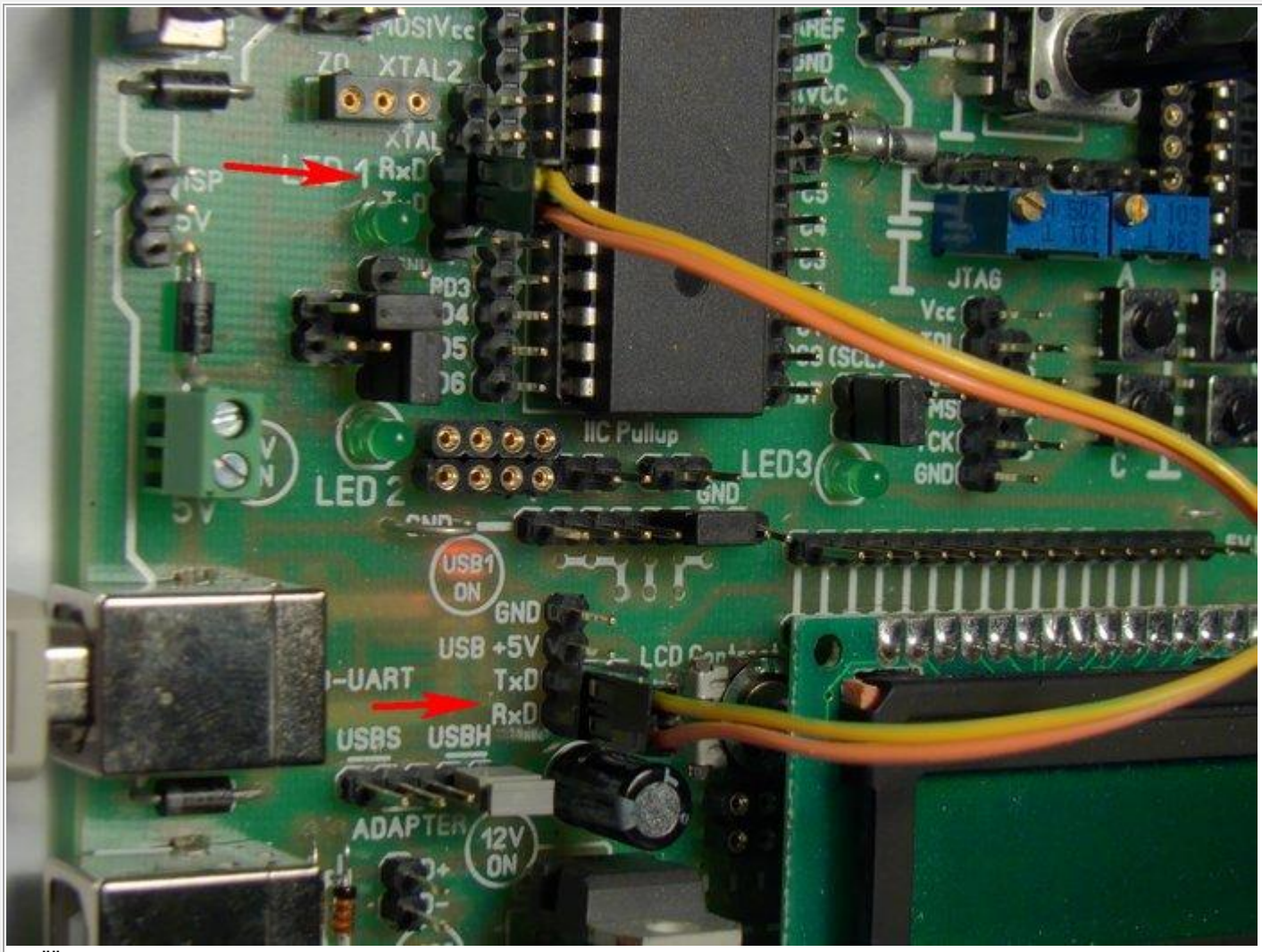


На этом пока все. Жми везде OK и переходи в исходник.

Постановка задачи
Чистый лист так и подмывает воплотить какую нибудь хитрую задумку, так как банальное мигание диодом уже не вставляет. Давай уж сразу брать быка за рога и реализуем связь с компом — это первым делом что я делаю.

Работать будет так:
При приходе по COM порту единички (код 0x31) будем зажигать диодик, а при приходе нуля (код 0x30) гасить. Причем сделано будет все на прерываниях, а фоновой задачей будет мигание другого диода. Простенько и со смыслом.

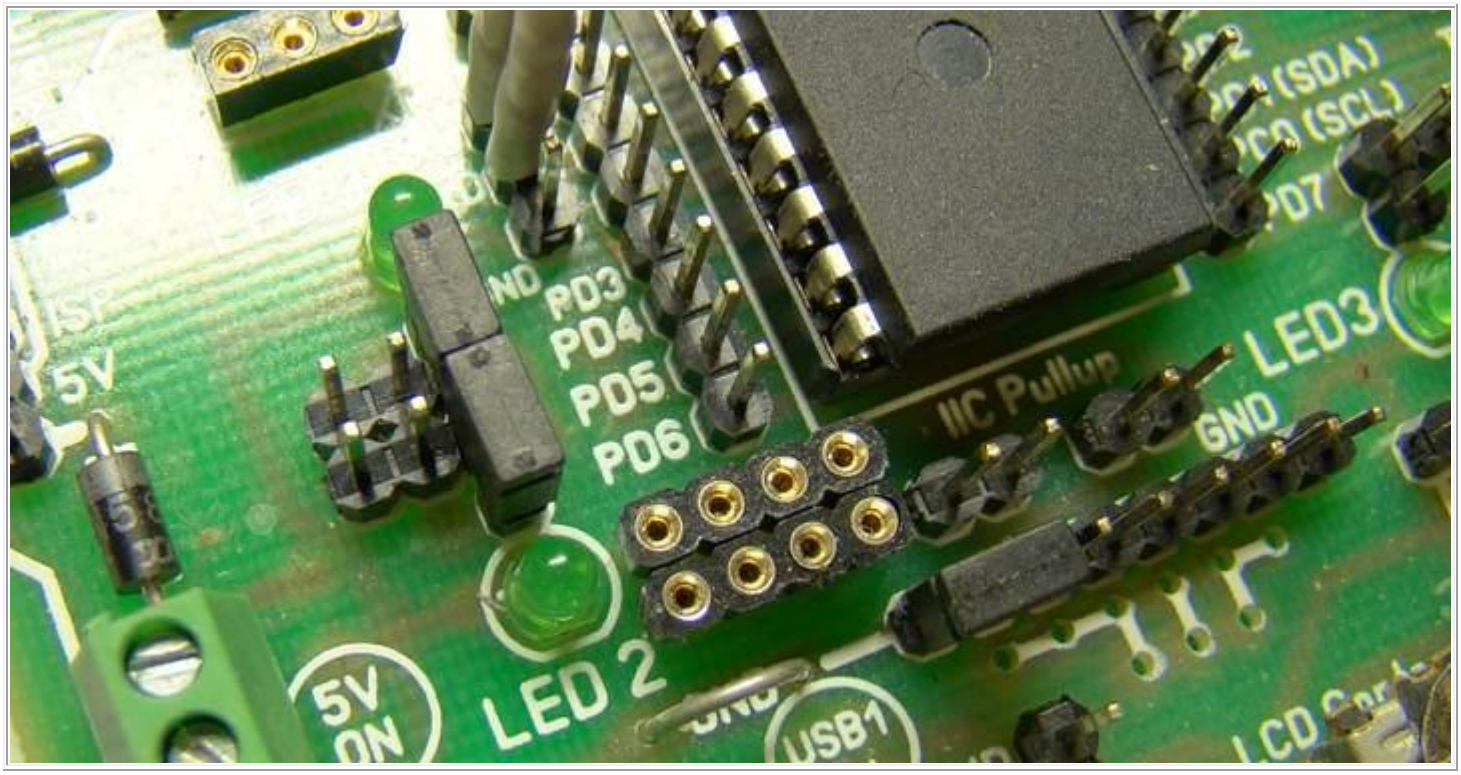
Собираем схему
Нам надо соединить модуль USB-USART конвертера с выводами USART микроконтроллера. Для этого берем перемычку из двух проводков и накидывам на штырьки крест накрест. То есть Rx контроллера соединяем с Tx конвертера, а Tx конвертера с Rx контроллера.



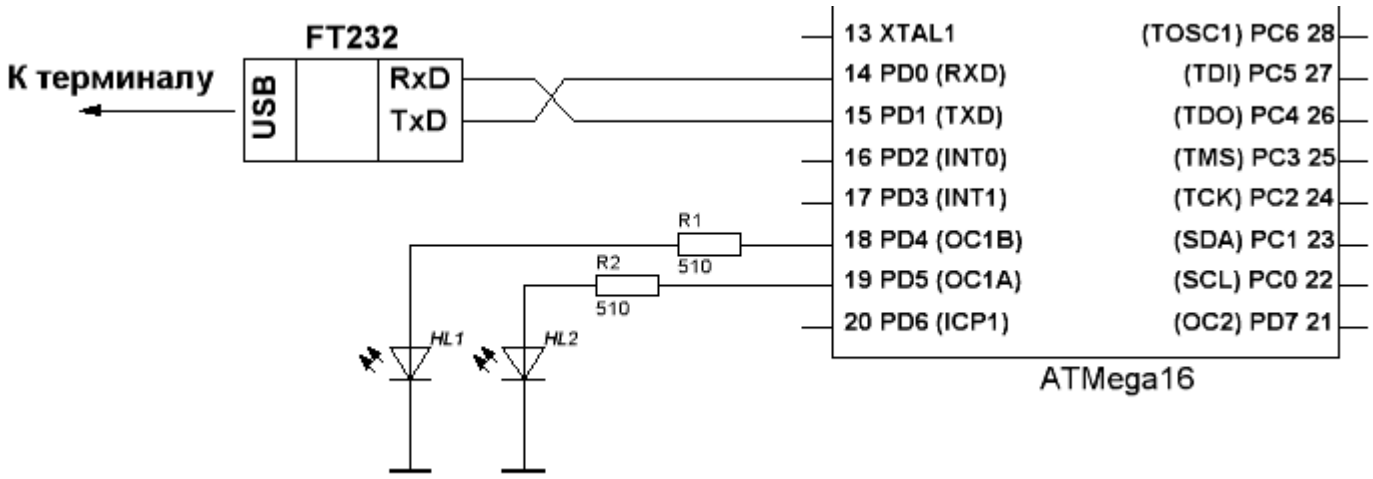
<=""">

Кроме того, через USART мы теперь сможем достучаться до загрузчика (Pinboard идет с уже прошитым загрузчиком) и прошить наш контроллер не используя программатор.

Также накинем джамперы, соединяющие LED1 и LED2. Тем самым мы подключим светодиоды LED1 и LED2 к выводам PD4 и PD5 соотверственно.



Получится, в итоге вот такая схема:



Подключение остальных выводов, питания, сброса не рассматриваю, оно стандартное

Пишем код

Сразу оговорюсь, что я не буду углубляться конкретно в описание самого языка Си. Для этого существует просто колоссальное количество материала, начиная от классики «Язык программирования Си» от K&R и заканчивая разными методичками.

Одна такая метода нашлась у меня в загашнике, я когда то именно по ней изучал этот язык. Там все кратко, понятно и по делу. Я ее постепенно верстаю и перестаскиваю на свой сайт.

Посмотреть оглавление [3].

Там правда еще не все главы перенесены, но, думаю, это ненадолго.

Вряд ли я опишу лучше, поэтому из учебного курса, вместо подробного разьяснения сишных тонкостей, я буду просто давать прямые линки на отдельные страницы этой методички.

Добавляем библиотеки.
Первым делом мы добавляем нужные библиотеки и заголовки с определениями. Ведь Си это универсальный язык и ему надо обьяснить что мы работаем именно с AVR, так что вписывай в исходник строку:

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

Этот файл находится в папке **WinAVR** и в нем содержится описание всех регистров и портов контроллера. Причем там все хитро, с привязкой к конкретному контроллеру, который передается компилятором через **make** файл в параметре **MCU** и на основании этой переменной в твой проект подключается заголовочный файл с описанием адресов всех портов и регистров именно на этот контроллер. Во как! Без него тоже можно, но тогда ты не сможешь использовать символические имена регистров вроде SREG или UDR и придется помнить адрес каждого вроде «0xC1”, а это голову сломать.

Сама же команда **#include <имя файла>** позволяет добавить в твой проект содержимое любого текстового файла, например, файл с описанием функций или кусок другого кода. А чтобы директива могла этот файл найти мы и указывали пути к нашему проекту (директория WinAVR там уже по дефолту прописана).

Главная функция.
Программа на языке Си вся состоит из функций. Они могут быть вложенными и вызываться друг из друга в любом порядке и разными способами. Каждая функция имеет три обязательных параметра:

- Возвращаемое значение, например, **sin(x)** возвращает значение синуса икс. Как в математике, короче.
- Передаваемые параметры, тот самый икс.
- Тело функции.

Все значения передаваемые и возвращаемые обязаны быть какого либо типа, в зависимости от данных.

Любая программа на Си должна содержать функцию **main** как точку входа в главную прогрмму, иначе это нифига не Си :). По наличию main в чужом исходнике из миллиона файлов можно понять, что это и есть головная часть программы откуда начинается все. Вот и зададим:

```
1 int main(void)
2 {
3
4 return 0;
5 }
```

Все, первая простейшая программа написана, не беда что она ничего не делает, мы же только начали.

Разберем что же мы сделали.
int это тип данных которая функция main возвращает. Узнать подробней о типах данных [4]

Конечно, в микроконтроллере **main** ничего вернуть в принципе не может и по идее должна быть **void main(void)**, но GCC изначально заточен на PC и там программа может вернуть значение операционной системе по завершении. Поэтому GCC на **void main(void)** ругается Warning’ом.

Это не ошибка, работать будет, но я не люблю варнинги.

void это тип данных которые мы передаем в функцию, в данном случае **main** также не может ничего принять извне, поэтом **void** — пустышка. Заглушка, применяется тогда когда не надо ничего передавать или возвращать.

Вот такие вот **{ }** фигурные скобочки это программный блок, в данном случае тело функции **main**, там будет распологаться код.

return — это возвращаемое значение, которое функция main отдаст при завершении, поскольку у нас int, то есть число то вернуть мы должны число. Хотя это все равно не имеет смысла, т.к. на микроконтроллере из main нам выходить разве что в никуда. Я возвращаю нуль. Ибо нефиг. А компилятор обычно умный и на этот случай код не

генерит.
Хотя, если извратиться, то из **main** на МК выйти можно — например вывалиться в секцию бутлоадера и исполнить ее, но тут уже потребуется низкоуровневое ковыряние прошивки, чтобы подправить адреса перехода. Ниже ты сам увидишь и поймешь как это сделать. Зачем? Вот это уже другой вопрос, в 99.999% случаев это нафиг не надо :)

Сделали, поехали дальше. Добавим переменную, она нам не особо нужна и без нужны вводить переменные не стоит, но мы же учимся. Если переменные добавляются внутри тела функции — то они локальные и существуют только в этой функции. Когда из функции выходишь эти переменные удаляются, а память ОЗУ отдается под более важные нужды. Подробней о локальных и глобальных переменных, а также времени жизни оных [5].

```
1 int main(void)
2 {
3 unsigned char i;
4
5 return 0;
6 }
```

unsigned значит беззнаковый. Дело в том, что в двоичном представлении у нас старший бит отводится под знак, а значит в один байт (char) влазит число +127/-128, но если знак отбросить то влезет уже от 0 до 255. Обычно знак не нужен. Так что **unsigned**.
i — это всего лишь имя переменной. Не более того.

Теперь надо проинициализировать порты и **UART**. Конечно, можно взять и подключить библиотеку и вызвать какой нибудь UartInit(9600); но тогда ты не узнаешь что же произошло на самом деле.

Делаем так:

```
1  int main(void)
2  {
3  unsigned char i;
4
5  #define XTAL 8000000L
6  #define baudrate 9600L
7  #define bauddivider (XTAL/(16*baudrate)-1)
8  #define HI(x) ((x)>>8)
9  #define LO(x) ((x)& 0xFF)
10
11 UBRRL = LO(bauddivider);
12 UBRRH = HI(bauddivider);
13 UCSRA = 0;
14 UCSRB = 1<<RXEN|1<<TXEN|1<<RXCIE|0<<TXCIE;
15 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
16 }
```

Страшна? На самом деле реалного кода тут всего пять последних строк. Все что **#define** это макроязык препроцессора. Почти та же ботва, что и в Ассемблере, но синтаксис несколько иной.

Они облегчат твои рутинные операции по вычислении нужных коэффициентов. В первой строке мы говорим что вместо **XTAL** можно смело подставлять 8000000, а **L**- указание типа, мол long — это тактовая частота процессора. То же самое **baudrate** — частота передачи данных по UART.

bauddivider уже сложней, вместо него будет подставлятся выражение вычисленное по формуле из двух предыдущих.
Ну, а **LO** и **HI** из этого результата возьмут младший и старший байты, т.к. в один байт оно явно может не влезть. В **HI** делается сдвиг икса (входной параметр макроса) восемь раз в вправо, в результате от него останется только старший байт. А в **LO** мы делаем побитовое И с числом 00FF, в результате останется только младший байт.

Так что все что сделано как **#define** можно смело выкинуть, а нужные числа подсчитать на калькуляторе и сразу же вписать их в строки UBBRL = …. и UBBRH = …..

Можно. Но! Делать этого **КАТЕГОРИЧЕСКИ НЕЛЬЗЯ**!

Работать будет и так и эдак, но у тебя в программе появятся так называемые **магические числа** — значения взятые непонятно откуда и непонятно зачем и если ты через пару лет откроешь такой проект то понять что это за значения будет чертовски трудно. Да и сейчас, захочешь ты изменить скорость, или поменяешь частоту кварца и все придется пересчитывать заново, а так поменял пару циферок в коде и все само. В общем, если не хочешь прослыть быдлокодером, то делай код таким, чтобы он легко читался, был понятен и легко модифицировался.

Дальше все просто:
Все эти «UBRRL и Co» это регистры конфигурации UART передатчика с помощью которого мы будем общаться с миром. И сейчас мы присвоили им нужные значения, настроив на нужную скорость и нужный режим.

Запись вида **1<<RXEN** Означает следующее: взять 1 и поставить ее на место **RXEN** в байте. **RXEN** это 4й бит регистра **UCSRB**, так что **1<<RXEN** образует двоичное число 00010000, **TXEN** — это 3й бит, а **1<<TXEN** даст 00001000. Одиночная «|» это побитовое **ИЛИ**, так что 00010000 | 00001000 = 00011000. Таким же образом выставляются и добавляются в общуюу кучу остальные необходимые биты конфигурации. В итоге, собраное число записывается в UCSRB. Подробней расписано в даташите на МК в разделе USART. Так что не отвлекаемся на технические детали.

Готово, пора бы посмотреть что получилось. Жми на компиляцию и запуск эмуляции (Ctrl+F7).

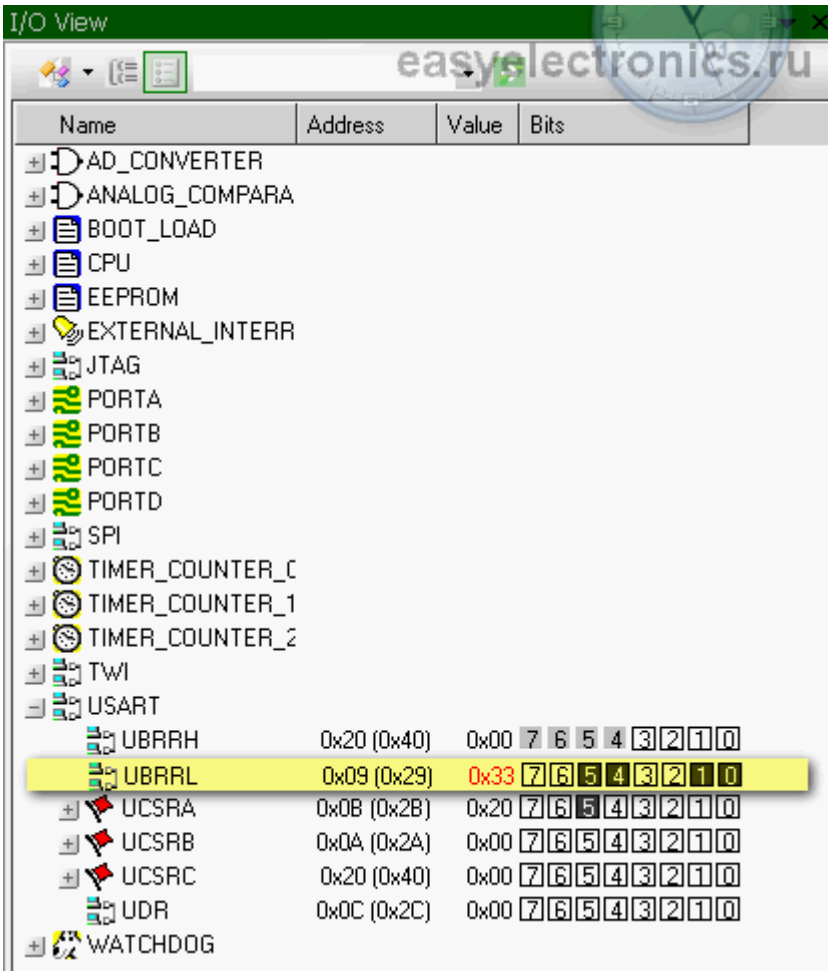
Отладка
Пробежали всякие прогресс бары, студия переменилась и возле входа в функцию main появилась желтая стрелочка. Это то где процессор в текущий момент, а симуляция на паузе.

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


int main(void)
{
unsigned char i;

#define XTAL 8000000L
#define baudrate 9600L
#define bauddivider (XTAL/(16*baudrate)-1)
#define HI(x) ((x)>>8)
#define LO(x) ((x)& 0xFF)

UBRRL = LO(bauddivider);
UBRRH = HI(bauddivider);
UCSRA = 0;
UCSRB = 1<<RXEN|1<<TXEN|1<<RXCIE|0<<TXCIE;
UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;

return 0;
}
```

Пора сделать ШАГ! Нажми F11!
Видишь стрелочка сразу же скакнула на вторую строку, на UBRRH = HI(bauddivider);

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


int main(void)
{
unsigned char i;

#define XTAL 8000000L
#define baudrate 9600L
#define bauddivider (XTAL/(16*baudrate)-1)
#define HI(x) ((x)>>8)
#define LO(x) ((x)& 0xFF)

UBRRL = LO(bauddivider);
UBRRH = HI(bauddivider);
UCSRA = 0;
UCSRB = 1<<RXEN|1<<TXEN|1<<RXCIE|0<<TXCIE;
UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;

return 0;
}
```

Дело в том, что изначально, на самом деле, она стояла на строке UBRRL = LO(bauddivider); Ведь то что у нас в define это не код, а просто предварительные вычисления, вот симулятор немного и затупил. Но теперь он осознал, первая инструкция выполнена и если ты залезешь в дерево **I/O View**, в раздел USART и поглядишь там на байт UBBRL то увидишь, что там значение то уже есть! 0x33.



Сделай еще один шаг. Погляди как изменится содержимое другого регистра. Так прошагай их все, обрати внимание на то, что все указаные биты выставляются как я тебе и говорил, причем выставляются одновременно для всего байта. Дальше Return дело не пойдет — программа кончилась.

Вскрытие
Теперь сбрось симуляцию в ноль. Нажми там **Reset (Shift+F5)**. Открывай дизассемблированный листинг, сейчас ты увидишь что происходит в контроллере в самом деле. **View -> Disassembler**. И не ЫЫАААА!!! Ассемблер!!! УЖОС!!! А НАДО. Чтобы потом, когда что то пойдет не так, не тупил в код и не задавал ламерских вопросах на форумах, а сразу же лез в потроха и смотрел где у тебя затык. Ничего там страшного нет.

Вначале будет ботва из серии:

```
1  +00000000:   940C002A    JMP       0x0000002A     Jump
2  +00000002:   940C0034    JMP       0x00000034     Jump
3  +00000004:   940C0034    JMP       0x00000034     Jump
4  +00000006:   940C0034    JMP       0x00000034     Jump
5  +00000008:   940C0034    JMP       0x00000034     Jump
6  +0000000A:   940C0034    JMP       0x00000034     Jump
7  +0000000C:   940C0034    JMP       0x00000034     Jump
8  +0000000E:   940C0034    JMP       0x00000034     Jump
9  +00000010:   940C0034    JMP       0x00000034     Jump
10 +00000012:   940C0034    JMP       0x00000034     Jump
11 +00000014:   940C0034    JMP       0x00000034     Jump
12 +00000016:   940C0034    JMP       0x00000034     Jump
13 +00000018:   940C0034    JMP       0x00000034     Jump
14 +0000001A:   940C0034    JMP       0x00000034     Jump
15 +0000001C:   940C0034    JMP       0x00000034     Jump
16 +0000001E:   940C0034    JMP       0x00000034     Jump
17 +00000020:   940C0034    JMP       0x00000034     Jump
18 +00000022:   940C0034    JMP       0x00000034     Jump
19 +00000024:   940C0034    JMP       0x00000034     Jump
20 +00000026:   940C0034    JMP       0x00000034     Jump
21 +00000028:   940C0034    JMP       0x00000034     Jump
```

Это таблица векторов прерываний. К ней мы еще вернемся, пока же просто посмотри и запомни, что она есть. Первая колонка — адрес ячейки флеша в которой лежит команда, вторая код команды третья мнемоника команды, та самая ассемблерная инструкция, третья операнды команды. Ну и автоматический коммент.
Так вот, если ты посмотришь, то тут сплошные переходы. А код команды JMP четырех байтный, в нем содержится адрес перехода, записанный задом наперед — младший байт по младшему адресу и код команды перехода 940C

Дальше идет предварительные приготовления, Си без них не может.

```
1 +0000002A:   2411        CLR       R1             Clear Register
```

Обнуление регистра R1

```
1 +0000002B:   BE1F        OUT       0x3F,R1        Out to I/O location
```

Запись этого нуля по адресу 0x3F, Если ты поглядишь в колонку I/O view, то ты увидишь что адрес 0x3F это адрес регистра SREG — флагового регистра контроллера. Т.е. мы обнуляем SREG, чтобы запустить программу на нулевых условиях.

```
1 +0000002C:   E5CF        LDI       R28,0x5F       Load immediate
2 +0000002D:   E0D4        LDI       R29,0x04       Load immediate
3 +0000002E:   BFDE        OUT       0x3E,R29       Out to I/O location
4 +0000002F:   BFCD        OUT       0x3D,R28       Out to I/O location
```

Это загрузка указателя стека. Напрямую грузить в I/O регистры нельзя, только через промежуточный регистр. Поэтому сначала LDI в промежуточный, а потом оттуда OUT в I/O. О стеке я тоже еще расскажу подробней. Пока же знай, что это такая динамическая область памяти, висит в конце ОЗУ и хранит в себе адреса и промежуточные переменные. Вот сейчас мы указали на то, откуда у нас будет начинаться стек.

```
1 +00000030:   940E0036    CALL      0x00000036     Call subroutine
```

Вот тут, собственно, идет переход к функции main. А через три команды как раз начинается Main. И переход идет через CALL, с сохранением адреса в стеке. При этом бездарно просерается два байта ОЗУ, а их всего 1024. =) Низачот! Впрочем, что то мне подсказывает, что обьявление main как **inline int main(void)** решит эту проблему, но не пробовал. Можешь сам проверить.

```
1 +00000032:   940C0041    JMP       0x00000041     Jump
```

Прыжок в сааааамый конец программы, а там у нас запрет прерываний и зацикливание наглухо само на себя:

```
1 +00000041:   94F8        CLI                      Global Interrupt Disable
2 +00000042:   CFFF        RJMP      PC-0x0000      Relative jump
```

Это на случай непредвиденых обстоятельств, например выхода из функции main. Из такого зацикливания контроллер можно вывести либо аппаратным сбросом, либо, что вероятней, сбросом от сторожевой собаки — watchdog. Ну или, как я говорил выше, подправить это мест в хекс редакторе и ускакать куда нам душе угодно. Также обрати внимание на то, что бывает два типа переходов JMP и RJMP первый это прямой переход по адресу. Он занимает четыре байта и может сделать прямой переход по всей области памяти. Второй тип перехода — RJMP — относительный. Его команда занимает два байта, но переход он делает от текущего положения (адреса) на 1024 шага вперед или назад. И в его параметрах указывается смещение от текущей точки. Используется чаще, т.к. занимает в два раза меньше места во флеше, а длинные преходы нужны редко.

```
1 +00000034:   940C0000    JMP       0x00000000     Jump
```

А это прыжок в самое начало кода. Перезагрузка своего рода. Можешь проверить, все вектора прыгают сюда. Из этого вывод — если ты сейчас разрешишь прерывания (они по дефолту запрещены) и у тебя прерывание пройзойдет, а обработчика нет, то будет программный сброс — программу кинет в самое начало.

Функция main. Все аналогично, даже можно и не описывать. Посмотри только что в регистры заносится уже вычисленное число. Препроцессор компилятора рулит!!! Так что никаких «магических» чисел!

```
1  +00000036:   E383        LDI       R24,0x33       Load immediate
2  +00000037:   B989        OUT       0x09,R24       Out to I/O location
3  15:       UBRRH = HI(bauddivider);
4  +00000038:   BC10        OUT       0x20,R1        Out to I/O location
5  16:       UCSRA = 0;
6  +00000039:   B81B        OUT       0x0B,R1        Out to I/O location
7  17:       UCSRB = 1<<RXEN|1<<TXEN|1<<RXCIE|0<<TXCIE;
8  +0000003A:   E988        LDI       R24,0x98       Load immediate
9  +0000003B:   B98A        OUT       0x0A,R24       Out to I/O location
10 18:       UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
11 +0000003C:   E886        LDI       R24,0x86       Load immediate
12 +0000003D:   BD80        OUT       0x20,R24       Out to I/O location
```

А вот тут косяк:

```
1 +0000003E:   E080        LDI       R24,0x00       Load immediate
2 +0000003F:   E090        LDI       R25,0x00       Load immediate
3 +00000040:   9508        RET                      Subroutine return
```

Спрашивается, для чего это компилятор добавляет такую ботву? А это не что иное, как Return 0, функцию то мы определили как int main(void) вот и просрали еще целых четыре байта не пойми на что :) А если сделать void main(void) то останется только RET, но появится варнинг, что мол у нас функция main ничего не возвращает. В общем, поступай как хошь :)

Сложно? Вроде бы нет. Пощелкай пошаговое исполнение в режиме дизассемблера и позырь как процессор выполняет отдельные инструкции, что при этом происходит с регистрами. Как происходит перемещение по командам и итоговое зацикливание.

Продолжение следует через пару дней …

А пока вот тебе готовый проект с тем, что есть сейчас [6]

Offtop:
Alexei78 сварганил плагинчик для файрфокса облегчающий навигацию по моему сайту и форуму. Обсуждение и скачивание, а также спасибы автору сего плагина в его теме на форуме [7]

AVR. Учебный Курс. Программирование на Си. Часть 2.

Вторая часть марлезонского балета, точнее введения в программирование на Си под микроконтроллеры.

Структура программы
Ну вот, МК у тебя уже кое что сделал. И хоть внешне ничего не видно, но внутри у него произошли изменения — передатчик стал готов к работе! Пора выдавать очередную порцию информации. Касательно того как вообще пишется программа для МК, не обязательно на Си. На чем угодно.

Компоновка любой программы такая:

```
1  Фунцкции
2   {
3   }
4
5  Прерывания
6   {
7   }
8
9  main()
10 {
11 инициализация;
12
13  Главный БЕСКОНЕЧНЫЙ цикл.
14    {
15     собственно программа
16    }
17 }
```

Инициализацию, по крайней мере UART мы уже сделали. Осталось добавить бесконечный цикл. Я люблю делать так:

```
1 while(1)
2 {
3 i++;
4 }
```

Просто и наглядно. **while** вначале проверяет условие в скобках, если оно не 0 то выполняет кодовый блок, потом возвращается обратно и проверяет снова. Ну а раз там по дефолту 1, то цикл будет бесконечным. Правда мне тут подсказывают, что по феншую кошерней будет сделать **for(; ;)**, мол так корректней. Не буду спорить, возможно они правы. Однако у меня **while(1)** никогда варнингов не давала и выглядит, ИМХО, логичней .

i++ это всего лишь увеличение i на единицу. Должны же мы в цикле что нибудь сделать. Не пустым же оставлять, да и переменная у нас была сделана по приколу.

Пускай тикает, а мы пока заглянем что появилось в дизассемблере.

Открываешь ты листинг, думаешь найти там новые ништяки… А ништяков то и нету!(с)

+0000003E: CFFF RJMP PC-0×0000 Relative jump

Вот это, наш while(1) и есть — глухое зацикливание.

Замечательно, а где же наша переменная i??? А это назвается восстание машин. Компилятор решил что он умней тебя и взял ее и похерил. В самом деле, решил он, нафига эта переменная вообще тут тикает? Полезной работы то от нее ноль! Нигде не применяется, нигде не сравнивается, нигде не используется. Только тикает тут себе в уголочке. Дай ка я ее грохну. И грохнул. Но мы то хотим чтобы тикало! Дать компилятору по башке и сказать кто тут главный можно. Для этого есть директива **volatile** — переменные с этой директивой оптимизатор не имеет права трогать.
Берем и приписываем ее нашей i, там где она определяется.

Получается:

```
1 volatile unsigned char i;
```

+00000043: 8189 LDD R24,Y+1 Load indirect with displacement
+00000044: 5F8F SUBI R24,0xFF Subtract immediate
+00000045: 8389 STD Y+1,R24 Store indirect with displacement
+00000046: CFFC RJMP PC-0×0003 Relative jump

Во, другой разговор. Взяли из памяти, вычли минус один (минус на минус дало плюс) и положили обратно где лежало. А потом зациклились.

Вот только что такое Y? Даташит нам скажет, что это регистровая пара R29:R28 и используется она для косвенной адресации. А откуда эта регистровая пара знает что там переменная «i»??? Начинаем штудировать код и в самом начале, где SREG обнуляется, находим:

+0000002C: E5CF LDI R28,0x5F Load immediate
+0000002D: E0D4 LDI R29,0×04 Load immediate

Теперь все понятно. Компилятор в самом начале, хитрый, занычил адрес Y, а потом заюзал его как только потребовалось.

Лепота!

Но пора бы добавить еще фишек. А конкретно диодиками мы там собирались мигать в фоновой программе. Оке, добавим в инициализацию порты диодов. На Pinboard все уже равзедено, поэтому там только перемычки поставить, что мы уже сделали.

Это будет PD4 и PD5. Единичка — горит, ноль не горит.

За настройку порта D отвечют регистры PORTD и DDRD — DDR это направление работы, а PORT состояние. Про порты и как их настраивать я уже писал ранее. [1]

Для зажигания LED1 нам надо выставить бит 4 в DDRD в единицу и дальше рулить битом 4 в регистре PORTD, для LED2 то же самое, только с битом 5

Так что добавляем после инициализации UART еще и инициализацию нашего порта

```
1 #define LED1 4
2 #define LED2 5
3 #define LED_PORT PORTD
4 #define LED_DDR DDRD
5 LED_DDR = 1<<LED1;
```

Можно было конечно просто написать DDRD = 0b00010000, но как я уже говори. **НИКАКИХ МАГИЧЕСКИХ ЧИСЕЛ!** Код должен быть понятным.

А вообще, все эти дефайны вообще по хорошему надо собрать в одну кучу и вынести в отдельный файл, подключив через include.

Захочешь все переделать — переделаешь только этот файл, а не будешь ковырять весь исходник. После, когда проект еще немного усложнится, мы это обязательно сделаем.

А наш цикл добавим команды установки и сброса бита в порту. Именно им будет определятся уровень напряжения на выходе.

```
1 while(1)
2  {
3  i++;
4  LED_PORT=0<<LED1;
5  LED_PORT=1<<LED1;
6  }
```

Во как! Если будешь гонять в отладчике, то увидишь как в I/O View будет менятся бит PORTD.4

Если заглянешь в дизасм, то увидишь как там хитро компилятор выкрутился — он при выполнении команды LED_PORT=1<<LED1;
Еще до входа в цикл загружает в регистр заранее число 0x10, а потом в цикле просто перекидывает его из регистра в порт. А ноль берет из R1 — R1 у нас по дефолту 0 всегда, это не закон, но прихоть компилятора.

32: LED_PORT=1<<LED1;
+00000045: E190 LDI R25,0×10 Load immediate

30: i++;
+00000046: 8189 LDD R24,Y+1 Load indirect with displacement
+00000047: 5F8F SUBI R24,0xFF Subtract immediate
+00000048: 8389 STD Y+1,R24 Store indirect with displacement

31: LED_PORT=0<<LED1;
+00000049: BA12 OUT 0×12,R1 Out to I/O location

32: LED_PORT=1<<LED1;
+0000004A: BB92 OUT 0×12,R25 Out to I/O location
+0000004B: CFFA RJMP PC-0×0005 Relative jump
32: LED_PORT=1<<LED1;

Прога получается целиком такой:

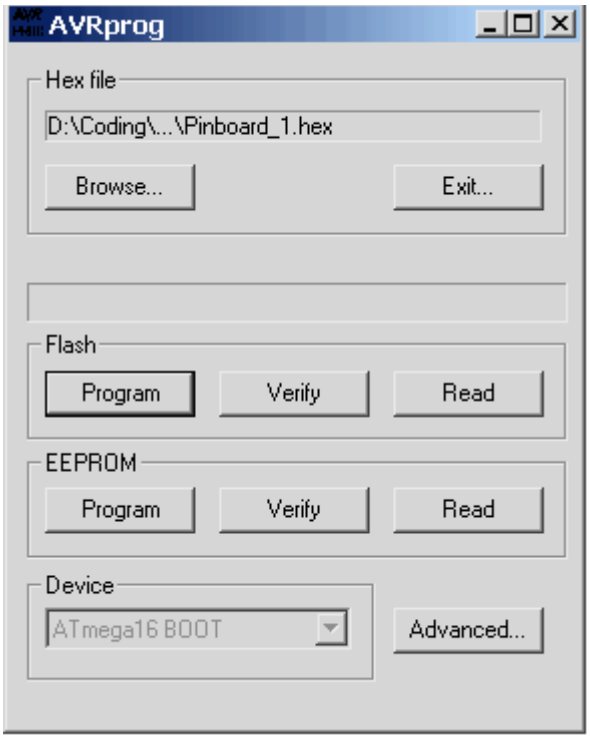
```
1  #include <avr/io.h>
2  int main(void)
3  {
4  volatile unsigned char i;
5
6  #define XTAL 8000000L
7  #define baudrate 9600L
8  #define bauddivider (XTAL/(16*baudrate)-1)
9  #define HI(x) ((x)>>8)
10 #define LO(x) ((x)& 0xFF)
11
12 UBRRL = LO(bauddivider);
13 UBRRH = HI(bauddivider);
14 UCSRA = 0;
15 UCSRB = 1<<RXEN|1<<TXEN|1<<RXCIE|0<<TXCIE;
16 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
17
18
19 #define LED1 4
20 #define LED2 5
21 #define LED_PORT PORTD
22 #define LED_DDR DDRD
23
```

```
24 LED_DDR = 1<<LED1;
25
26 while(1)
27 {
28 i++;
29 LED_PORT=0<<LED1;
30 LED_PORT=1<<LED1;
31 }
32
33 return 0;
34 }
```

Пора пихнуть ее в реальный контроллер и посмотреть что будет.

Тыкай в квадратную кнопку Reset и пока горит LED2 тут же запускай AVR PROG. Он должен определить бут и сказать, что плата есть (подробная инструкция в картинках как заливать прошивку через бутлоадер PinBoard [2] есть в четвертой части документации из рассылки).

Заливай получившийся hex файл.



Жми RESET на плате и наблюдай что происходит.

Те же у кого моей демоплаты [2] нет — выкручивайтесь как хотите :) Сразу могу подсказать, что вам потребуется программатор. Например, такой [3] или такой [4].

На PinBoard после сброса картина будет такой:
Вначале загорится LED2 — это работает BootLoader. Через пять секунд она погаснет, но загорится наш LED1.

Вот только какого черта он горит? Он ведь должен вроде бы мигать! Диод мигать то будет, но вот уж больно быстро он будет это делать.
Если сейчас тыкнуться осцилом в ножку PD4 то увидишь такую картину:

Узкий провал это интервал между

```
1 LED_PORT=0<<LED1;
2 LED_PORT=1<<LED1;
```

Широкий — весь остальной цикл. Нетрудно увидеть разницу исполнения кода по времени. =)

Но нам то надо мигание! Надо бы замедлить. Очевидно что между включением и выключением надо воткнуть задержку.

Пока я не буду тебя грузить программированием задержек. Поэтому подключай хидер библиотеки **delay.h**Можешь поискать ее в потрохах **WinAVR**

```
1 #include <avr/delay.h>
```

В самое начало засунь, в компанию к уже имеющемуся.

Добавил? Сразу же скомпиль. Опа, а там два варнинга.
Варнинги это хуже ошибок! Забьешь на них — замучаешься отлаживать!

Смотрим что там есть:
c:/winavr-20090313/lib/gcc/../../avr/include/avr/delay.h:36:2: warning: #warning «This file has been moved to .»
Ну это фигня. Компилятор нас предупреждает, что разработчики WinAVR поместили Delay.h в другую папку, а в папке /avr осталась затычка стрелочник. Но лучше подправить, чтобы не мозолила глаз

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

Остался другой варнинг:
c:/winavr-20090313/lib/gcc/../../avr/include/util/delay.h:85:3: warning: #warning «F_CPU not defined for «

Это посереьзней. Компилятор говорит нам, что переменная препроцессора F_CPU не определена.

Длительность задержки зависит от частоты процессора, а частоту то мы не указали. Не вопрос! Добавляй куда нибудь в начало, но обязательно ПЕРЕД

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

Иначе тебе еще три варнига будет — компилятор читает программу сверху вниз, а именно в delay.h ему потребовалась эта переменная. А как же частота которую мы выставляли в настройках проекта? А она видимо передается через командную строку, но вот delay требует ее раньше чем она передается в проект. Поэтому добавим в код.

У меня теперь начало вот так:

```
1 #define F_CPU 8000000L
2 #include <avr/io.h>
3 #include <util/delay.h>
```

Да, помнишь мы там дефайнили XTAL? Мне лень было переписывать старый добрый макрос, а тебе будет полезно. Чтобы было однообразие в коде, замени все XTAL на F_CPU — ближе к стандарту.

Теперь компилятор ругается на то, что у него F_CPU два раза встречается:

../Pinboard_1.c:1:1: warning: «F_CPU» redefined

Не беда, заходим в свойства проекта и выносим оттуда нашу частоту, оставляя пустое поле.

Все, теперь не ругается. Без ошибок и варнингов компилится.

Потрошим хидеры
Библиотеку то мы добавили, а как ей пользоваться? Можно, конечно поискать мануал на **WinAVR**, но этот путь не для нас. Настоящие джедаи не пользуются мануалами, а вырывают информацию силой! По крайней мере на этапе обучения. Тяжело в учении легко в бою!(С)

Есть библиотека, надо узнать что у ней внутри. Сама библиотека эта в виде скомпилированого и заоптимизированного наглухо бинарика, а описание ее функций находится в **delay.h** лезь туда и не пугайся кучи барахла что там есть. Тебе надо найти прототип функции.

В Си нужно четко указать компилятору параметры функции (что на входе, имя, и что на выходе) перед тем как он ее встретит в тексте. Делается это либо посредством прототипа — просто копия заголовока функции без тела, либо просто саму функцию пишут перед всем кодом (обычно так, ибо писать прототипы всем лень).

Поэтому то часто Сишные программы кверх ногами пишутся — главный цикл в самой заднице. Но не рекомендую делать так — это типичный пример быдлокодерства, т.к. читать такую программу неудобно.

Лезь в **delay.h** и ищи в этой помойке что либо похожее на прототип или функцию. Да, октрывай этот файл не блокнотом, а в самой студии — будет подсветка кода. Как только ты его заинклюдил то он сразу же появится в дереве проекта. Ты должен нарыть что нибудь вроде:

```
1 static inline void _delay_us(double __us)
2 static inline void _delay_ms(double __ms)
```

Как видишь, в этой либе всего две фунции **delay_us** и **delay_ms**, дедуктивный метод мне подсказывает, что одна это выдержка в миллисекундах, а вторая в микросекундах. А в качестве передаваемого параметра будет число формата double.

Отрывай от нее все причиндалы, они нужны только на этапе описания, и вставляй в код, не забыв указать задержку. Вот так вот:

```
1 while(1)
2 {
3 i++;
4 LED_PORT=0<<LED1;
5  _delay_ms(1000);
6 LED_PORT=1<<LED1;
7  _delay_ms(1000);
8 }
```

Погасили, подождали секунду, зажгли, подождали секунду, перешли в начало цикла.

Скомпилили, прошили, нажали RESET, дождались пока бутлоадер перейдет к исполнению… Мигает. На глазок примерно 1 секунда.

Ок, надо бы проверить вообще то у нас получилось или не то? Задержка точно выдерживается?

В студии есть вкладка **Processor**, а в ней есть графа **StopWatch** — это время работы процессора. По ней можно вычислять как долго работает та или иная функция. Изначально значения там в микросекундах, что не удобно. Тыкни на нее мышой и переключи в миллисекунды. 1000мс = 1 секунда.

Чуть выше тактовая частота процессора, на которой идет симуляция. Наверняка там 4Мгц стоит :), а у нас 8. Это не страшно, зайди потом как нибудь во время активного процесса отладки в меню Debug ->AVR Simulator options и выбери там нужную частоту.

Теперь дошагай до строки перед задержкой, а потом поставь курсор после задержки и сделай «выполнить до курсора» (Ctrl+F10) После чего, студия задумается и спустя какое то время проскочит через задержку,а в графе StopWatch будет пройденное время.

Но куда прикольней в этот момент наблюдать содержимое дизассемблера. Видел какая каша в коде на Си? Куча барахла, условий, из одной библиотеки вызывается другая. А на асме, в реале:

—- c:\WinAVR\avr\include\util\delay_basic.h ——————————————————
105: __asm__ volatile (
+00000045: EC28 LDI R18,0xC8 Load immediate
+00000046: E030 LDI R19,0×00 Load immediate
Загружаем первую половину задержки 0xC800

= тут немного другого кода — компилятор порой размазывает функции по всему исходнику =

+0000004C: E180 LDI R24,0×10 Load immediate
+0000004D: E297 LDI R25,0×27 Load immediate
Загружаем вторую половину задержки 0×2710

+0000004E: 01F9 MOVW R30,R18 Copy register pair
+0000004F: 9731 SBIW R30,0×01 Subtract immediate from word
+00000050: F7F1 BRNE PC-0×01 Branch if not equal
—- c:\WinAVR\avr\include\util\delay.h ——————————————————
124: __ticks —;
+00000051: 9701 SBIW R24,0×01 Subtract immediate from word
120: while(__ticks)
+00000052: F7D9 BRNE PC-0×04 Branch if not equal

А вот и сама задержка. Если приглядеться и скинуть комментарии, которые только в глаза лезут, то выглядит это так, в псевдокоде:

```
1          R18_R19 = 0xC800               //Первая половина задержки - база
2          R25_R24 = 0x2710               // Вторая половина задержки - задержка
3
4
5  M0:     R30_R31 = R18_R19          ----------------------
6  M1:     R30_R31 - 1                          |        |
7          R30_R31 равно нулю?                  |        |
8          Нет = переход на метку M1 -----------         |
9                                                        |
10         R25_R24 -1                                    |
11         R30_R31 равно нулю?                           |
12         Нет = переход на метку M0       ----------------
```

Как видишь, получается цикл двойной вложенности. В результате внутренний цикл выполняется 0xC800*0×2710 раз. Казалось бы, возможная выдержка в таком случае составляет 0xFFFF*0xFFFF итераций — ведь в первом цикле слово декрементируется и во втором цикле слово. А вот нифига! Си требует совместимости и универсальности либ. В данном случае на входе ВРЕМЯ, а функция оперирует только числом итераций, поэтому для любой тактовой частоты должно быть четко отмерено время. Поэтому значения базовой загрузки хитро вычисляются еще на этапе компиляции препроцессором Си и подставляются в функцию уже заточенные под конкретное значение кварца.

Так какая же максиальная выдержка у этой фунции? А стоит порыться в том же delay.h чтобы найти ответ:

The maximal possible delay is 262.14 ms / F_CPU in MHz.

When the user request delay which exceed the maximum possible one, _delay_ms() provides a decreased resolution functionality. In this mode _delay_ms() will work with a resolution of 1/10 ms, providing delays up to 6.5535 seconds (independent from CPU frequency). The user will not be informed about decreased resolution.

А по русски — 262.14 миллисекунд/частоту в мегагерцах. Это если в точном режиме, а если надо больше, то идет уменьшение дискретности, в итоге на точности 1/10 миллисекунды мы можем выжать из нее максимум 6.5535 секунды вне зависимости от частоты кварца.

Стоить запомнить эту ситуацию на будущее, если вдруг будешь писать «Универсальную библиотеку на все случаи жизни». Чтобы не получить вилы из-за переполнения потому что кварц не тот :)

Кстати, обрати внимание в прототипе на директиву **inline** — это значит, что функция будет не вызываться, а тупо подставляться в код. Словно макрос. Два раза написали мы инлайновую функцию delay_ms — получили два экземпляра этой функции в коде. Можешь сам убедиться. Обычно это не очень оптимально с точки зрения расхода ПЗУ, но зато работает быстрей. Впрочем, сам же видишь что функция тут пара команд. Так что, ИМХО, inline тут более чем оправдана — вызов ее потребует передачу параметра double, сохранение и восстановление адреса возврата и прочий сопутствующий код, а это будет не намного короче.

Как обычно, проект на данной стадии. [5]

Продолжение следует….

З.Ы.
Не пугайтесь что продвижение идет в час по чайной ложке, у меня на этот счет шкала логарифмическая :) А пока вжевывайте в то как работает контроллер на простых примерах, потому что когда в бой пойдет многозадачность и разделение процессов отвлекаться на фигню будет некогда :)

AVR. Учебный Курс. Программирование на Си. Часть 3

Фоновую программку мы сделали, интерфейс связи с компом инициализировали, надо бы сделать так, чтобы наш контроллер мог принимать сигналы от компа. Проще всего это сделать на прерываниях.

Что такое прерывание?
Прерывание это аппаратное событие, например, байт пришел в порт, на выводе изменился логический уровень, АЦП обсчитала напряжение или таймер дотикал до переполнения. В общем, любой аппаратный сигнал. Когда сигнал приходит, то периферийный блок в своем регистре поднимет флаг прерывания.

Когда приходит прерывание то контроллер завершит текущую команду (машинную инструкцию!) сразу же кинется выполнять процедуру обработки прерывания, а как выполнит, то вернется к прерваной фоновой программе.

Прерывания можно, а часто необходимо запрещать, чтобы посреди критичного участка не ускакать выполнять невесть что. Запрещать их можно глобально, флагом I в регистре SREG, а можно локально — запрещая источник каждого прерывания индивидуально. По дефолту, при сбросе, все прерывания от устройств запрещены, глобальный флаг тоже сброшен. Включем мы их по мере надобности.

Поскольку прерывание приходит ВНЕЗАПНО, а у нас могут быть несохраненные данные, то обработчик их должен сохранить и при выходе в фоновую программу вернуть все как было.

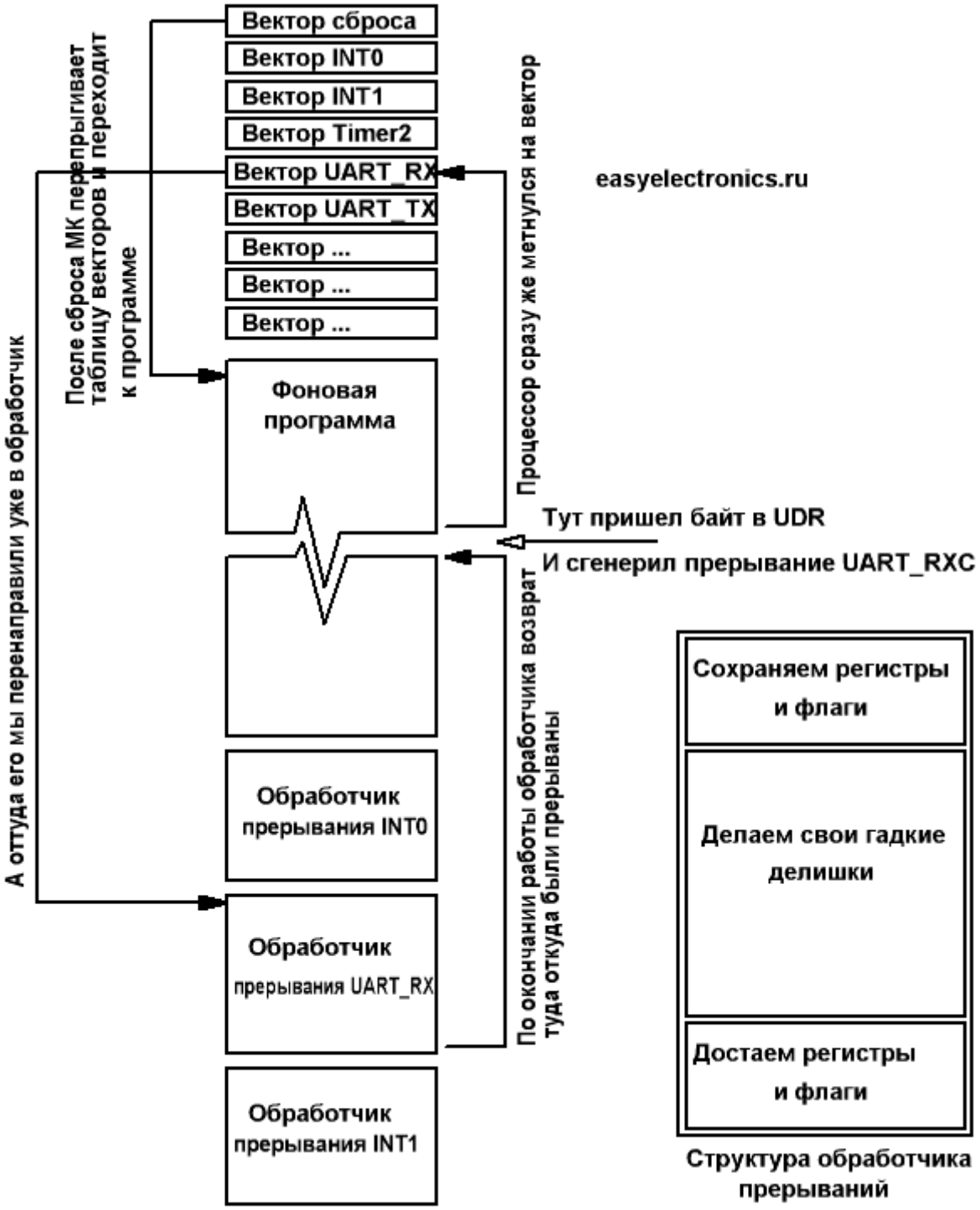
Впрочем, если бездумно подходить к этому делу, то можно огрести адские хаотичные глюки. Особенно при использовании высокоуровневых языков вроде Си, где вся эта процедура скрыта от глаз программиста и если он не волокет в асме и не понимает работу контроллера на уровне машинных инструкций, то ошибку найти не сможет. Но об этом чуть позже, когда буду расписывать отладку и ошибки.

Вектора прерываний

Как процессор узнает куда ему перепрыгивать? А по вектору! Помнишь я тебе показывал на таблицу векторов прерываний? Она в самом начале памяти идет. Вот это оно.

Вектор это адрес перехода. У каждого аппаратного события имеющего прерывание есть свой вектор. Аппаратных событий у AVR тьма, поэтому таблица прерываний весьма толстая, десятки адресов.

Когда происходит прерывание, то ядро контроллера запоминает текущий адрес в стеке и делает прыжок на адрес вектора соответствующего прерывания. И все. Дальше наша забота — в ячейку памяти на которую указывает вектор прерывания мы вписываем свою команду RJMP которая перенаправит проц на реальный код обработчика.



Приоритеты прерываний
Тут принцип просто — кто раньше встал того и тапки. Когда процессор уходит на обработку прерывания, то аппаратно происходит запрет всех остальных прерываний.

Его, конечно, можно включить программно, установив флаг I в регистре SREG и тогда у нас будут вложенные прерывания, но обращатся с этим следует ОЧЕНЬ осторожно. Так как множественные прерывания нагружают стек и может произойти его переполнение, что даст полный сбой работы и начнется невесть что. Полная упячка.

По выходу из обработчика, по команде RETI флаг I вернется в прежнее состояние.

У многих сразу возникнет вопрос, а что будет если во время обработки одного прерывания придет другое? Оно потеряется?

Нет, не потеряется — у него взведется флаг и как только мы завершим первый обработчик автоматом произойдет переход к отложенному прерыванию. Единственное, что мы можем потерять количество одинаковых прерываний. Т.е. если обрабатывается, например, INT0 и прерывания запрещены, а в это время придет три раза INT1, то на выходе INT1 обработается только один раз.

А если во время обработки первого прерывания случится не одно, а, скажем, два или три? Какое из этих отложеных прерываний будет выполнено первым? А по порядку в таблице векторов. У кого адрес младше тот и пойдет первым.

Теперь тезисно:

- Прерывания это аппаратные события.
- У **каждого** прерывания есть **свой** персональный адрес вектор, по которому и будет послан проц в случае чего.
- По дефолту они запрещены локально и глобально.
- Вызов прерывания может быть когда угодно и где угодно, между любыми двумя командами (хотя хрена там, между CLI CLI его не будет :))
- В обработчике прерывания надо учитывать тот факт, что данные нужно сохранять и восстанавливать при выходе.
- Приоритет прерываний работает по принципу «кто первый встал тот и ходит в тапках». Остальные запрещены аппаратно, но можно разрешить программно уже в обработчике.
- Проснувшиеся после никуда не теряются и ждут своих тапок — права порулить процом, в порядке жесткой очереди по таблице прерываний.
- Прерывания это колоссальный источник глюков и головная боль любого быдлокодера. Но без них никак, поэтому эту тему надо знать вдоль и поперек.
- Посколькоу прерывание ВНЕЗАПНОЕ и может быть где угодно, то обработчик прерывания должен выполняться МАКСИМАЛЬНО КОРОТКО И БЫСТРО. Зашел, отметился, вышел. Только так. Никаких задержек и длинных циклов. Все сложные вещи только в фоновой задаче. Но об этом подробней позже.

Перечитать три раза. Запомнить навсегда.

Когда пишешь на Си, то прервания можно использовать только по назначению. Ассемблерщики же могут извращаться, например, для создания уберточного кода когда нужно выдерживать длительность с точностью до такта. .

Итак, теорию повторили, вернемся к практике.

Прерывание у нас будет от USART. Пришел байт — вызвалось прерывание.

Раз мы решили заюзать прерывания, то сначала надо подключить библиотеку прерываний:

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

Оформляется прерывание в WinAVR как:

```
1 ISR(вектор прерывания)
2 {
3 }
```

Так что берем и добавляем в код, где нибудь до или после процедуры main эту бодягу. Вот только как узнать что вписать в вектор прерывания? Мануалы к черту — считаем что их нет. Инфу будем добывать раскопками, выгрызая из подножного корма.

Прерывания вещь интимная и зависят от конкретной модели контроллера. Так что искать описание векторов прерываний надо искать в файле который описывает наш контроллер.

Какой это файл? А позырь в дерево проектов, ветка зависимостей. Что там? У меня Mega16 и там есть файл iom16.h Из всех файлов он больше всех похож на искомый — потому как только он явно для меги16 =).

Там куча всего, что искать? А все что связано с прерываниями — ищи Interrupt и все что рядом.

Очень скоро найдешь секцию /* Interrupt vectors */ и там будут все вектора, в том числе и на USART

```
1  /* Interrupt vectors */
2  /* Vector 0 is the reset vector. */
3  /* External Interrupt Request 0 */
4  #define INT0_vect                    _VECTOR(1)
5  #define SIG_INTERRUPT0               _VECTOR(1)
6
7  /* External Interrupt Request 1 */
8  #define INT1_vect                    _VECTOR(2)
9  #define SIG_INTERRUPT1               _VECTOR(2)
10
11 /* Timer/Counter2 Compare Match */
12 #define TIMER2_COMP_vect             _VECTOR(3)
13 #define SIG_OUTPUT_COMPARE2          _VECTOR(3)
14
15 /* Timer/Counter2 Overflow */
16 #define TIMER2_OVF_vect                      _VECTOR(4)
17 #define SIG_OVERFLOW2                _VECTOR(4)
18
19 /* Timer/Counter1 Capture Event */
20 #define TIMER1_CAPT_vect             _VECTOR(5)
21 #define SIG_INPUT_CAPTURE1           _VECTOR(5)
22
23 /* Timer/Counter1 Compare Match A */
24 #define TIMER1_COMPA_vect            _VECTOR(6)
25 #define SIG_OUTPUT_COMPARE1A         _VECTOR(6)
26
27 /* Timer/Counter1 Compare Match B */
28 #define TIMER1_COMPB_vect            _VECTOR(7)
29 #define SIG_OUTPUT_COMPARE1B         _VECTOR(7)
30
31 /* Timer/Counter1 Overflow */
32 #define TIMER1_OVF_vect                      _VECTOR(8)
33 #define SIG_OVERFLOW1                _VECTOR(8)
34
35 /* Timer/Counter0 Overflow */
36 #define TIMER0_OVF_vect                      _VECTOR(9)
37 #define SIG_OVERFLOW0                _VECTOR(9)
38
39 /* Serial Transfer Complete */
40 #define SPI_STC_vect                 _VECTOR(10)
41 #define SIG_SPI                      _VECTOR(10)
42
43 /* USART, Rx Complete */
44 #define USART_RXC_vect               _VECTOR(11)
45 #define SIG_USART_RECV               _VECTOR(11)
46 #define SIG_UART_RECV                _VECTOR(11)
47
48 /* USART Data Register Empty */
49 #define USART_UDRE_vect                      _VECTOR(12)
```

```
50 #define SIG_USART_DATA                 _VECTOR(12)
51 #define SIG_UART_DATA                  _VECTOR(12)
52
53 /* USART, Tx Complete */
54 #define USART_TXC_vect                 _VECTOR(13)
55 #define SIG_USART_TRANS                        _VECTOR(13)
56 #define SIG_UART_TRANS                 _VECTOR(13)
57
58 /* ADC Conversion Complete */
59 #define ADC_vect                       _VECTOR(14)
60 #define SIG_ADC                        _VECTOR(14)
61
62 /* EEPROM Ready */
63 #define EE_RDY_vect                    _VECTOR(15)
64 #define SIG_EEPROM_READY               _VECTOR(15)
65
66 /* Analog Comparator */
67 #define ANA_COMP_vect                  _VECTOR(16)
68 #define SIG_COMPARATOR                 _VECTOR(16)
69
70 /* 2-wire Serial Interface */
71 #define TWI_vect                       _VECTOR(17)
72 #define SIG_2WIRE_SERIAL               _VECTOR(17)
73
74 /* External Interrupt Request 2 */
75 #define INT2_vect                      _VECTOR(18)
76 #define SIG_INTERRUPT2                 _VECTOR(18)
77
78 /* Timer/Counter0 Compare Match */
79 #define TIMER0_COMP_vect               _VECTOR(19)
80 #define SIG_OUTPUT_COMPARE0            _VECTOR(19)
81
82 /* Store Program Memory Ready */
83 #define SPM_RDY_vect                   _VECTOR(20)
84 #define SIG_SPM_READY                  _VECTOR(20)
85
86 #define _VECTORS_SIZE 84
```

Прорва, на каждый чих, но нам сейчас интересны те прерывания которые отвечают за USART

Вот они, все три. Одна завершение приема, вторая опустошение регистра ака готовность к передаче следующего байта, третья завершение передачи.

```
1  /* USART, Rx Complete */
2  #define USART_RXC_vect                 _VECTOR(11)
3  #define SIG_USART_RECV                 _VECTOR(11)
4  #define SIG_UART_RECV                  _VECTOR(11)
5
6  /* USART Data Register Empty */
7  #define USART_UDRE_vect                        _VECTOR(12)
8  #define SIG_USART_DATA                 _VECTOR(12)
9  #define SIG_UART_DATA                  _VECTOR(12)
10
11 /* USART, Tx Complete */
12 #define USART_TXC_vect                 _VECTOR(13)
13 #define SIG_USART_TRANS                        _VECTOR(13)
14 #define SIG_UART_TRANS                 _VECTOR(13)
```

Мы собрались по входному сигналу делать экшн. Так что применяй первый. Какой из трех? Без разницы, они равнозначны и их тут столько исключительно из совместимости с разными версиями компиляторов.

Вписываешь в код вот такую шнягу:

```
1 ISR(USART_RXC_vect)
2 {
3
4 }
```

Осталось прописать в код то что мы должны сделать из прерывания. Первое — нужно обязательно считать данные из регистра UDR иначе флаг прерывания останется висеть и оно будет непрерывно вызываться еще и еще.

Для определения байта можно применить конструкцию Switch-case
В итоге, получилась такая вещь:

```
1 ISR(USART_RXC_vect)
2 {
3 switch(UDR)
4         {
5         case '1': LED_PORT = 1<<LED2;  break;
6         case '0': LED_PORT = 0<<LED2;  break;
7         default: break;
8         }
9 }
```

Как видишь, у нас в свитче берется UDR и в зависимости от того чему он равен, символу нуля или символу единицы осуществляется переход. Если не то и не другое, то переход на default и сразу выход из свитча — break. break это вообще выход из программной структуры. Например, если сделаешь break в цикле, то выпадешь из цикла.

В значении case можно было бы написать и код символа, было бы case 0×31: но, как я говорил, никаких цифр. Если есть возможность их избежать — избегай всеми силами. Пусть за тебя препроцессор компилятора отдувается.

В частности ‘символ’ это ASCII код символа. Удобно!

Да, обрати внимание на то, что в каждой строке case стоит break. Это неспроста! Если break там не будет, то пройдя на case 0 и выполнив там все процессор перейдет к следующему case и так далее, пока не выйдет из него совсем или не нарвется на break.

Что зажигания диодов, то тут, как видишь, пришлось добавить еще и LED2, прописав его в дефайнах. И не забыв проинициализировать его порт на выход! В итоге, стало выглядеть так:

```
1  LED_DDR = 1<<LED1|1<<LED2;
```

Правда сразу ничего у меня не заработало — посыпалось куча ошибок. Пришлось взять все дефайны в кучу и поднять их над всем кодом — компилятор же читает код сверху вниз так что вначале он должен прочитать все макроопределения прежде чем ему встретятся операции с ними.

Вот текущий код целиком:

```
1  #define F_CPU 8000000L
2  #define LED1           4
3  #define LED2           5
4  #define LED_PORT       PORTD
5  #define LED_DDR        DDRD
6
7
8  #include &lt;avr/io.h>
9  #include <util/delay.h>
10 #include <avr/interrupt.h>
11
12 ISR(USART_RXC_vect)
13 {
14 switch(UDR)
```

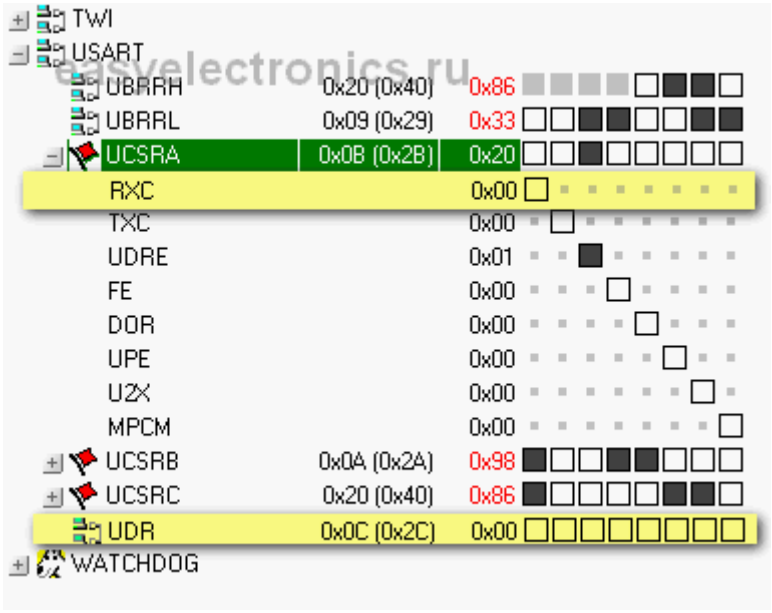
```
15         {
16         case '1': LED_PORT = 1<<LED2;  break;
17         case '0': LED_PORT = 0<<LED2;  break;
18         default: break;
19         }
20 }
21
22 int main(void)
23 {
24 volatile unsigned char i;
25
26 #define XTAL 8000000L
27 #define baudrate 9600L
28 #define bauddivider (XTAL/(16*baudrate)-1)
29 #define HI(x) ((x)>>8)
30 #define LO(x) ((x)& 0xFF)
31
32 UBRRL = LO(bauddivider);
33 UBRRH = HI(bauddivider);
34 UCSRA = 0;
35 UCSRB = 1<<RXEN|1<<TXEN|1<<RXCIE|1<<TXCIE;
36 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
37
38
39 LED_DDR = 1<<LED1|1<<LED2;
40
41
42 while(1)
43  {
44  i++;
45  LED_PORT=0<<LED1;
46  _delay_ms(1000);
47  LED_PORT=1<<LED1;
48  _delay_ms(1000);
49  }
50
51 return 0;
52 }
```

Вроде все написано. Запускай код на эмуляцию. Но вот как проверить прерывание?

В данном случае, в студии, для этого нет никаких эмуляторов терминала (вообще есть hapsim, а еще можно отлаживать в Proteus или VMLAB, но об этом я пожалуй расскажу попозже).

Но можно сделать вид, что пришло прерывание, словно терминал сработал. **Процессор определяет, что произошло прерывание когда периферия, его генерирующая, поднимет флаг прерывания**.

У USART на прием флаг RxC вот возьми и ткни его, мол прерывание свершилось.



Вручную выстави этот бит, найдя его в окне i/o view в ветви USART. Поставил, а потом сделай пару шагов по F11 и… нифига не произошло!

А почему? а потому, что мы забыли эти прерывания врубить. Частая ошибка начинающих. Дело в том, что мало разрешить прерывания при инициализации UART — это мы разрешили локальные прерывания, уартовские, а есть еще флаг глобального разрешения — флаг I он по дефолту сброшен.

Установка и сброс флага разрешения глобальных прерываний делается командой **SEI** и **CLI** соотвественно. В Си есть макрос sei(); и cli(); который просто подставит эту ассемблерную команду. Берем и добавляем его в конце инициализации:
Закопипащу немного кода оттуда, чтобы ты понял куда я его сунул:

```
1 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
2 LED_DDR = 1<<LED1|1<<LED2;
3 sei();                          // <--- вот оно.
4
5 while(1)
```

Перезапусти эмуляцию и снова ткни флаг, сделав после пару шагов.

Во, теперь у тебя программа должна будет оказаться в ISR стоя на заголове Switch. Пора проверить как работает.

В этот момент открой **I/O View** и натыкай в регистре UDR число 0x31 или просто ткни в окошко со значением и введи там код 0x31.

Нажми F11 опа! мы перешли к первой строке case и зажгли диодик, а после сразу же вышли из кейса и из прерывания вообще.

Вскрытие прерывания
Теперь пора показать тебе как работает прерывание изнутри. Переходи в дизассемблер и смотри что там поменялось. В первую очередь, изменился переход на векторе номер 11 — вектор RXC — мы ведь на него обработчик повесили.

….
+00000014: 940C0034 JMP 0×00000034 Jump
+00000016: 940C0036 JMP 0×00000036 Jump
+00000018: 940C0034 JMP 0×00000034 Jump
….

А со строки

@00000036: __vector_11

начинается наш обработчик. Поскольку прерывание может вызваться в любой момент, где угодно, то вначале надо сохранить все регистры, что мы будем использовать в обработчике прерывания. В том числе регистр флагов. Сохраняются они в стек и выглядит это как пачка команд PUSH

```
1 +00000036:    921F        PUSH      R1              Push register on stack
2 +00000037:    920F        PUSH      R0              Push register on stack
3 +00000038:    B60F        IN        R0,0x3F         In from I/O location
4 +00000039:    920F        PUSH      R0              Push register on stack
5 +0000003A:    2411        CLR       R1              Clear Register
6 +0000003B:    938F        PUSH      R24             Push register on stack
```

По такой бодяге легко отличить обработчик прерывания. То что там затесался IN R0,0x3F это не случайно — таким образом идет сохранение регистра SREG в котором хранятся флаги. Их тоже сохранять обязательно, ведь от них зависят операции условий.

Дальше идет наш switch:
Свитч отличить тоже несложно — обычно это куча последовательных сравнений — CPI

```
1    switch(UDR)
2 +0000003C:    B18C        IN        R24,0x0C           //Берем значение из UDR
3 +0000003D:    3380        CPI       R24,0x30     // И сравниваем его последовательно с
4 кодами
5 +0000003E:    F029        BREQ      PC+0x06    // И переход на обработку если совпало
6 +0000003F:    3381        CPI       R24,0x31     // Аналогично и далее
  +00000040:    F421        BRNE      PC+0x05
```

И по очереди сравниваем с кодами 0x30 и 0×31 в случае совпадения или не совпадения осуществляем переход либо на default либо на нужный case: которые будут ниже по тексту:

```
1         case '1': LED_PORT = 1<<LED2;  break;
2 +00000041:    E082        LDI       R24,0x02       Load immediate
3 +00000042:    BB88        OUT       0x18,R24       Out to I/O location
4 +00000043:    C001        RJMP      PC+0x0002      Relative jump
5 17:             case '0': LED_PORT = 0<<LED2;  break;
6 +00000044:    BA18        OUT       0x18,R1        Out to I/O location
7 20:        }
```

Прерывание завершается массовым доставанием из стека данных которые мы туда сунули и командой RETI
Данные достаются в обратно порядке!

```
1 5:    918F        POP       R24             Pop register from stack
2 +00000046:    900F        POP       R0              Pop register from stack
3 +00000047:    BE0F        OUT       0x3F,R0         Out to I/O location
4 +00000048:    900F        POP       R0              Pop register from stack
5 +00000049:    901F        POP       R1              Pop register from stack
6 +0000004A:    9518        RETI
```

Работает! Задание вроде мы выполнили. Но …

Я оставил в ней кучу идиотских багов и ляпов. В педагогических целях. Чтобы было что отлаживать в следующей части. Там мы их найдем и уничтожим — ты запомнишь ряд характерных ошибок на которых осекаются новички, а я покажу ряд приемов по отладке.

Оставайтесь на линии :)

Текущее состояние программы в виде проекта для AVRStudio. [1]

AVR. Учебный Курс. Программирование на Си. Часть 4

Теперь глянем на нашу программу, скомпилим, прошьем, поглядим как выполняется.

Зашиваю все через AVR Prog в Pinboard и смотрю на поведение LED1 и LED2.

LED1 мигает как и задумано, но стоит мне попытаться зажечь LED2 отправкой с терминала «1”, как первый диод гаснет. И наоборот — зажженый диод LED2 гаснет вместе с первым. Бага! Причем жирная такая. Рассмотрим откуда она взялась.

Вот код мигания первым диодом:

```
1 LED_PORT=1<<LED1;
2 _delay_ms(1000);
3 LED_PORT=0<<LED1;
4 _delay_ms(1000);
```

А вот код работы с вторым диодом:

```
1 switch(UDR)
2         {
3         case '1': LED_PORT = 1<<LED2; break;
4         case '0': LED_PORT = 0<<LED2; break;
5         default: break;
6         }
```

Как видишь, тут мы пишем в один и тот же порт, но вот только биты разные. Но нельзя вот так просто через операцию «=» изменить один бит! (только если мы используем битовые поля, о них я расскажу позже) [1]. Так что операция идет с целым байтом, и в LED_PORT поочередно записывается число 00100000 (1<<LED2) и 00010000 (1<<LED1), перезаписывая друг друга. Поэтому когда происходит запись одного значения мы теряем прердыдущее. А 0<<LED2 это по факту просто 0, потому что как ноль по байту не двигай нулем он и останется .

Особенно часто эта ошибка всплывает у начинающих когда они пытаются инициализировать периферию не всю сразу, а по мере надобности. Включил, например, прерывания от таймера 1, а когда активизировал таймер 2, то затер биты таймера1. Как итог — результат не соответствует ожиданиям.

Как быть? Тут нам помогут битовые маски. Помнишь логические операции AND/OR/NOT/XOR?

Вот на их основе и будет работа. Маски накладываются побитно. Так что

	Установка бита	Сброс бита	Инверсия байта	Инверсия отдельных бит
Исходное значение	10001000	10001000	10001000	10001000
Операция	**OR (|)**	**AND (&)**	**NOT (~)**	**XOR (^)**
Битовая маска	00010000	01111111	n/a	11000000
Результат	10011000	00001000	01110111	01001000

Так что применяя на исходный байт битовую маску (байт с установленными в нужном порядке битами) мы можем сбросить, установить, или инвертировать любой бит, а также сразу инвертировать весь байт.

Вот только не следует путать поразрядные операции (например &) с логическими (&&). [2]Первые действую бит против бита, вторые сразу целиком весь байт, анализируя его по принципу равено он нулю или нет. Если в байте будет хоть одна единица, то он уже не равен нулю.

Так что теперь операция установки бита будет выглядеть так:

```
1 LED_PORT = LED_PORT | 1<<LED2;
```

Т.е. берем предыдущее значение порта LED_PORT и накладываем на него маску 1<<LED2 которая установит биты. Результат помещаем обратно где взяли.

Для сброса бита нам маску надо инвертировать, поэтому выглядеть будет так:

```
1 LED_PORT = LED_PORT & ~(1<<LED2);
```

Согласно приоритету выполнения [3] операций вначале выполняется выражение в скобках (битмаска), потом оно инвертируется, а затем накладывается на значение порта. И все записывается в прежнее значение.

Ну и для инверсии бита (0 меняется на 1, а 1 на 0) используется такая конструкция:

```
1 LED_PORT = LED_PORT ^ (1<<LED2);
```

Также Си допускает сокращенную конструкцию подобной записи логических операций:

```
1 LED_PORT &= ~(1<<LED2);
2 LED_PORT |= 1<<LED2;
3 LED_PORT ^=1<<LED2;
```

Так что смело правим нашу прогу получается вот такая конструкция:

```
1 switch(UDR)
2         {
3         case '1': LED_PORT |= 1<<LED2;         break;
4         case '0': LED_PORT &= ~(1<<LED2);      break;
5         default: break;
6         }
```

и

```
1 while(1)
2  {
3 i++;
4 LED_PORT ^=1<<LED1;
5 _delay_ms(1000);
6  }
```

Так как нам надо просто мигать, то достаточно инвертировать бит, и код становится еще меньше. Мне тут с галерки подсказывают, что в новых моделях AVR для того, чтобы инвертировать бит достаточно записать в регистр PIN бит на соответствующую ногу. Не знаю, не проверял. А в даташите на атмега 168 (один из новых) регистр PIN обозначен как READ ONLY.

Сами операции выглядят довольно минималистично:
Инверсия через чтение-модификацию-запись

```
1 46:       LED_PORT ^=1<<LED1;
2 +00000060:   B382        IN        R24,0x12       In from I/O location
3 +00000061:   2784        EOR       R24,R20        Exclusive OR
4 +00000062:   BB82        OUT       0x12,R24       Out to I/O location
```

А работа с портами через специальные команды установки и сброса битов.

```
1 17:            case '1': LED_PORT |= 1<<LED2;         break;
2 +00000041:   9A95        SBI       0x12,5         Set bit in I/O register
```

```
1         case '0': LED_PORT &= ~(1<<LED2);     break;
2 +00000043:   9895        CBI       0x12,5         Clear bit in I/O register
```

Но этот прикол нашли все кто внимательно в код позырил. А есть более тонкие глюки. В частности я тут «бездумно» скопипастил код инициализации USART из другого проекта.

```
1  #define XTAL 8000000L
2  #define baudrate 9600L
3  #define bauddivider (XTAL/(16*baudrate)-1)
4  #define HI(x) ((x)>>8)
5  #define LO(x) ((x)& 0xFF)
6
7  UBRRL = LO(bauddivider);
8  UBRRH = HI(bauddivider);
9  UCSRA = 0;
10 UCSRB = 1<<RXEN|1<<TXEN|1<<RXCIE|1<<TXCIE;
11 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
```

Казалось бы все хорошо, работает. Где тут бага?

А все дело в прерывании. Мы создали обработчик для USART_RXC_vect, а для TX ничего не писали, хотя вектор прерывания от передатчика включен. Да, в данный момент нам это не мешает — т.к. мы ничего не передаем, но программы имеют тенденцию усложняться и кто знает во что это выльется. А что будет если произойдет прерывание которого нет? А давай поглядим на вектора:

```
1  +00000000:   940C002A    JMP       0x0000002A     Jump
2  +00000002:   940C0034    JMP       0x00000034     Jump
3  +00000004:   940C0034    JMP       0x00000034     Jump
4  +00000006:   940C0034    JMP       0x00000034     Jump
5  +00000008:   940C0034    JMP       0x00000034     Jump
6  +0000000A:   940C0034    JMP       0x00000034     Jump
7  +0000000C:   940C0034    JMP       0x00000034     Jump
8  +0000000E:   940C0034    JMP       0x00000034     Jump
9  +00000010:   940C0034    JMP       0x00000034     Jump
10 +00000012:   940C0034    JMP       0x00000034     Jump
11 +00000014:   940C0034    JMP       0x00000034     Jump
12 +00000016:   940C0036    JMP       0x00000036     Jump
13 +00000018:   940C0034    JMP       0x00000034     Jump
14 +0000001A:   940C0034    JMP       0x00000034     Jump
15 +0000001C:   940C0034    JMP       0x00000034     Jump
16 +0000001E:   940C0034    JMP       0x00000034     Jump
17 +00000020:   940C0034    JMP       0x00000034     Jump
18 +00000022:   940C0034    JMP       0x00000034     Jump
19 +00000024:   940C0034    JMP       0x00000034     Jump
20 +00000026:   940C0034    JMP       0x00000034     Jump
21 +00000028:   940C0034    JMP       0x00000034     Jump
22 +0000002A:   2411        CLR       R1             Clear Register
23 +0000002B:   BE1F        OUT       0x3F,R1        Out to I/O location
24 +0000002C:   E5CF        LDI       R28,0x5F       Load immediate
25 +0000002D:   E0D4        LDI       R29,0x04       Load immediate
26 +0000002E:   BFDE        OUT       0x3E,R29       Out to I/O location
27 +0000002F:   BFCD        OUT       0x3D,R28       Out to I/O location
28 +00000030:   940E004A    CALL      0x0000004A     Call subroutine
29 +00000032:   940C006B    JMP       0x0000006B     Jump
30 +00000034:   940C0000    JMP       0x00000000     Jump
```

Как видишь, все пустые вектора ведут на метку 00000034, где нас ждет

```
1 +00000034:   940C0000    JMP       0x00000000     Jump
```

Это переход на вектор сброса.

Т.е. случайно разрешенное, но неопределенное прерывание вызовет программный RESET. Вот забудешь такую мелочь, а потом будешь ловить причину сброса проверяя питание и наводки :)

Хотя такой маневр компилятора относительно неопределенных векторов у меня вызывает недоумение. Дело в том, что при писании прог на ассемблере принято глушить вектора не переходами в ноль, а командами RETI — в этом случае прерывание как вызывалось так и убралось. И программа потеряла только несколько тактов, а не работосопособность.

С другой стороны вешать туда сброс это максимизация ошибки. Т.е. большую багу когда не работает вообще все отловить легче.

Еще есть в этом коде еще мощные вилы связанные с неатомарностью (атомарный доступ — в одну команду, без возможности прерывания) доступа к регистрам периферии и памяти. Но об этом я расскажу в следующий раз.

Анонс:
Прерывание + доступ к памяти = головная боль быдлокодера.
Архитектура построения программы контроллера — то о чем дружно забыли составители курсов.
Отладочные приемы или как эффективно травить клопов в матрасе

AVR. Учебный Курс. Программирование на Си. Атомарные операции.

Не раз и не два сталкивался с утверждением, что изучать ассемблер микроконтроллера это всего лишь пустая трата времени, дескать все можно сделать на Си, а если сильно надо то комманды можно и в даташите поглядеть.

Сейчас я одним маленьким примерчиком это утверждение зарою в землю, а сверху накрою могильной плитой.

Итак, есть у нас такой код (не ищите в нем практического смысла, я его просто как пример работы с разными операндами написал):

```
1  volatile char flag_byte;
2  /*Просто флаговая переменная, на разные случаи жизни. Разные события там
3  выставляют флажки, опираясь на которые потом работает логика программы.
4  Один из способов организации псевдомногозадачности. Когда у нас главный цикл
5  анализирует флажки и делает переходы на подпрограммы, а вызов подпрограмм
6  осуществляется не напрямую, а установкой соответствующих флажков. Своего
7  рода диспетчер переходов. О такой архитектуре я скоро расскажу)*/
8
9  ISR (USART_RXC_vect)    // Обработчик прерывания, самый обычный.
10 {
11 flag_byte|=1<<rcv_buff;
12 ...
13 ...
14 }
15
16 int main (void)         // Главная программа
17 {
18 INIT_ALL();
19 SEI();
20 ...
21 ...
22 ...
23 TCCR0A          |=1<<WGM01;
24 ...
25 flag_byte       |=1<<options;
26 ...
27 PORTB    &=~(2<<1);
28 ...
29 }
```

Человек ни разу не писавший под МК на ассемблере, начавший изучать МК сразу с Си, схавает такую конструкцию и даже не поморщится. Прожженый ассемблерщик же нецензурно матюгнется, обзовет первого быдлокодером и исправит исходник следующим образом:

```
volatile char flag_byte;       // Просто флаговая переменная под флаги на разные случаи жизни

ISR (USART_RXC_vect)    // Обработчик прерывания
{
flag_byte|=1<<rcv_buff;
...
...
}

int main (void)         // Главная программа
{
INIT_ALL();
SEI();
...
...
...
TCCR0A  |=1<<WGM01;
...
CLI();
flag_byte |=1<<options;
SEI();
...
PORTB &=~(2<<1);
...
}
```

А еще проверит не юзается ли где еще в прерываниях регистр TCCR0A. И внимательно посмотрит на работу с регистрами периферии.

В чем же дело? Какая разница между

```
TCCR0A  |=1<<WGM01;
flag_byte |=1<<options;
PORTB &=~(1<<2);
```

Или между

```
PORTB &=~(1<<2|1<<3);
```

и

```
PORTB &=~(1<<3);
```

Для программиста сишника в принципе никакой — и там и там какой то регистр или переменная. Единственное что это аппаратный регистры и поэтому могут меняться самопроизвольно, да переменная в прерывании меняется, поэтому идет с квалификатором **volatile**.

А для ассемблерщика между этими строками заложена огромная разница.

Во первых flag_byte расположена в памяти и это флаговый регистр, а изменение бита в памяти может быть только через чтение-модификацию-запись. Соответственно

```
flag_byte |=1<<options;
```

Компилируется во что то вроде:

```
1         LDS     R16,flag_byte   ; Чтение
2         ORI     R16,1<<options  ; Модификация
3         STS     flag_byte,R16   ; Запись
```

Во первых **TCCR0A** это аппаратный регистр, причем имеющий адрес в пределах **00-3F** (адресное пространство на котором возможна работа команд **IN/OUT**), поэтому доступ к нему может быть через команду **OUT**, причем напрямую в регистр **I/O** сделать **OUT** константы нельзя, только через промежуточный регистр **R16…R31** так что конструкция

```
1 TCCR0A  |=1<<WGM01;
```

Будет подобна предыдущей:

```
1         IN      R16,TCCR0A      ; Чтение
2         ORI     R16,1<<WGM01    ; Модификация
3         OUT     TCCR0A,R16      ; Запись
```

А вот регистр **PORTB** расположен в адресном пространстве от **00…1F**, поэтому кроме команды **OUT** до него могут дотянуться также команды работы с битами **CBI** и **SBI**.

Так что конструкция

```
1 PORTB &=~(1<<2);
```

Скомпилируется в

```
1         CBI     PORTB,2
```

А вот :

```
1 PORTB &=~(1<<2|1<<3);
```

Может уже быть как

```
1         CBI     PORTB,2
2         CBI     PORTB,3
```

так и

```
1         IN      R16,PORTB               ; Чтение
2         ANDI    R16,~(1<<2|1<<3)        ; Модификация
3         OUT     PORTB,R16               ; Запись
```

В зависимости от числа одновременно меняемых битов и умности компилятора.

Да, это все здорово. Но к чему я это?

А к тому, что у нас тут есть еще и прерывание. В котором может быть изменены наши значения, а еще тот милый факт что прерывание, будучи разрешенным, может выскочить между двух любых инструкций. И при этом родить очень адский глюк которой может ВНЕЗАПНО вылезти через пару лет безглючной работы, а потом также бесследно исчезнуть, оставив разработчика задумчиво чесать репу на предмет «ЧТО ЭТО БЫЛО»? Часто тут грешат на глюки железа и дырявые процы. Хотя на самом то деле просто программа была корявая.

Покажу на примере, вот наш первоначальный код:

```
volatile char flag_byte;       // Просто флаговая переменная под флаги на разные случаи жизни
ISR (USART_RXC_vect)    // Обработчик прерывания
{
flag_byte|=1<<rcv_buff;
...
...
}

int main (void)         // Главная программа
{
INIT_ALL();
SEI();
...
...
...
TCCR0A  |=1<<WGM01;
flag_byte |=1<<options;
PORTB &=~(2<<1);
...
}
```

Программа выполняется себе, готовится записать бит options в переменную flag_byte…

```
; начало операции flag_byte |=1<<options;
        LDS     R16,flag_byte   ; Чтение предыдущего значения
```

А тут хопа, так фаза Луны совпала и вдруг пришло прерывание именно в этот момет. А что у нас в прерывании? Правильно! Сохранение регистров, что то вида::

```
; Начало обработчика вектора ISR (USART_RXC_vect)
        PUSH    R17
        IN      R17,SREG
        PUSH    R17
        PUSH    R16
```

А дальше наше тело прерывания:

```
; Начало операции flag_byte|=1<<rcv_buff;
        LDS     R16,flag_byte   ; Чтение
        ORI     R16,1<<rcv_buff ; Модификация
        STS     flag_byte,R16   ; Запись
; Конец операции flag_byte|=1<<rcv_buff;
```

Выставили мы битик **rcv_buff** и записали в ту же переменную флагов. А что потом? Правильно — возврат регистров из стека и возврат:

```
        POP     R16
        POP     R17
        OUT     SREG,R17
        POP     R17
        RETI
; Конец прерывания ISR (USART_RXC_vect)
```

Куда возврат? Туда же где прервались:

```
        ORI     R16,1<<options  ; Модификация
        STS     flag_byte,R16   ; Запись
; конец операции flag_byte |=1<<options;
```

Замечательно, а что же у нас в регистре R16 в данный момент?

Очевидно то же самое, что и до входа в прерывание — состояние флаговой переменной flag_byte. Вот только заковыка — **это состояние флаговой переменной уже не актуально, устарело!** Т.к. его только что изменило прерывание, выставив там какой то другой свой флаг. И теперь, произведя модификацию и запись у нас в **flag_byte** будет записана не актуальная инфа уже с двумя стоящими флагами rcv_buff и options, а восстановленная из «бэкапа» предыдущая копия с одним лишь **options**. А событие которое нам пыталось донести прерывание, выставив свой флаг, было забыто вообще.

И что самое гадкое такие ошибки фиг отловишь если не знаешь что такое может произойти. И могут они быть сразу, а могут коварно переждать в засаде весь процесс отладки, а когда девайс уйдет в серийное производство или на ответственный пост внезапно выскочат и хорошо если никто не умрет.

Лирическое отступление, можно пропустить:
Когда то давно отлаживал я программу одну, она была на ассемблере, но там я просто забыл что у меня есть переменная — адрес перехода. Которая может меняться и в прерывании тоже. Так вот, все работало идеально, но примерно на 30 минуте работы программа вставала колом или начинала гнать полную ересь. Я убил тогда на отладку около недели. Программа была довольно большая, там могло быть что угодно, начиная от переполнения стека до срыва очередей диспетчера задач. В ход пошла уже тяжелая артиллерия в лице логического анализатора и JTAG — без толку. Попробуй поймай багу на хрен знает какой итерации главного цикла, да еще при совпадении условий внешних воздействий в фиг знает каком порядке. Трассировать или отлаживать в протеусе это почти бесполезно.
Причем малейшие изменения в коде приводили к видоизменению баги. Добавил в код парочку NOP — клиническая картина резко меняется. Чудом мне удалось поймать момент, когда прогу перекашивало точно на 25 итерации главного цикла после отпускания кнопки. Выглядело это примерно так: послать в USART точно 10 байт, причем последний должен быть непременно «R», потом нажать кнопку и не отпускать пока не будет нажата вторая. И вот когда вторую отпустить прога встает колом.

Дотрассировав до этого места в JTAG я наконец понял где собака порылась — адрес перехода состоял из двух байт и лежал в ОЗУ. Первый байт менялся в фоне программы, и мог быть изменен в прерывании. И вот когда это прерывание вклинивалось в между первым и вторым байтом при занесением нового адреса, то в результате получался гибрид состоящий из двух разных половинок адреса, ясен фиг что переход получался черти куда и выглядело это как срыв стека, хотя стек тут был вообще не причем. Впрочем, про характерную клиническую картину ошибок я тоже собираюсь потом написать.

Итак, враг найден. Но что же с ним делать? А делать тут собственно нечего. Единственный способ избежать этой напасти это либо не использовать раздельно используемые переменные, либо запрещать прерывания при их модификации в фоновой программе — делать атомарными.

Атомарная — значит не делимая. Не делимая прерыванием. Поэтому то команды запрета/разрешения прерываний

```
1 cli();
2 flag_byte |=1<<options;
3 sei();
```

Спасают ситуацию.

А остальные? Они ведь тоже делаются не за одну операцию! Ну в данном случае в прерывании они не используются и поэтому пока можно спать спокойно, но держать на контроле надо.

А вот операции вроде SBI/CBI делаются за одну команду и являются атомарными от природы.

Страшно да? А ведь многие об этом даже не задумывались.

Да и в самом деле, что считать атомарным а что нет? Не закрывать же в конце концов параноидально все операции с разделяемыми переменными и IO регистрами командами cli/sei. Как никак прерывания вещь важная и созданная именно для того, чтобы реагировать быстро.
И, например, на AVR нельзя сделать атомарную операцию на регистры ввода вывода старше 1F — там просто нет команд логических операций на IO, а вот на PIC24 (а может и на более младших, хз не знаю я PIC) можно, через битовые маски по XOR, т.к. там есть команда xor по IO. Подробней можно прочитать в [одной замечательной статье](#) [1], она хоть и про PIC24, но полезна будет всем.

Так что мало знать что есть вилы, надо уметь их обходить с минимальными потерями. Вот для этого то и надо хорошо знать ассемблер того контроллера на котором пишешь, на уровне хотя бы десятка — двух программ отличных от тупой мигалки светодиодом. Чтобы сразу за сишным кодом видеть возможные ассемблерные инструкции, знать что будет делать компилятор в том или ином случае. Где можно ожидать вилы в стоге сена.

Отлично, враг известен, чем его мочить тоже выяснили. Но любое оружие может замочить и хозяина. Если им неумело пользоваться. Поясню на примере:

Вот сделали мы какую нибудь функцию, а чтобы прерывания нам не нагадили мы добавили в них конструкцию cli/sei на критичные места. Все довольны, все счастливы. Ага, до тех пор пока в целях оптимизации и универсальности кода мы не вкатим эту функцию в обработчик прерывания. В принципе, ничего страшного нет, пользоваться в прерываниях функциями можно. Вот только надо учитывать что в прерываниях прерывания же аппаратно запрещены, а наша функция по выходу из критических мест их разрешает. И тут у нас вылезают другие вилы — вложенные прерывания. Это бага не столь законспирированная как неатомарный доступ, но вылезти тоже может ой как не сразу.

Что делать? А тут все просто. Если у нас есть функция с атомарными операциями и юзается она и в прерываниях и в основном теле программы, то надо бы это учитывать и не менять флаг прерывания зря. Т.е. если он был сброшен (в прерывании) то мы его и не возвращаем обратно — незачем.

На ассемблере это может выглядеть, например, так:

```
1  ; Begin Atomic Block
2          IN      R17,SREG        ; Сохранили регистр SREG, а в нем и флаг I
3          PUSH    R17             ; можно в стеке, можно еще где. Не принципиально
4          CLI                     ; Запретили прерывания
5
6          ...                     ; Тут у нас код который должен быть атомарным
7          ...                     ;
8          ...                     ;
9
10         POP     R17             ; Достали сохраненное в стеке
11         OUT     SREG,R17        ; Вернули SREG на место
12 ; End Atomic Block
```

Разрешать прерывания тут не надо. Т.к. мы сохранили сразу весь **SREG**. Если прерывания были разрешены, то по доставании его из стека они достанутся разрешенными. Ну а если не были разрешены то ничего и не изменится.

На Си же можно тупо в лоб проверять условием наличие флага **I** регистра **SREG** и в зависимости от этого делать потом разрешение прерывания или нет. Запрещать их в любом случае. Либо применить инлайновый ассемблер своего Си компилятора.

А еще можно поискать уже готовые макросы в составе компилятора.

В WinAVR GCC например есть такой хидер как **util/atomic.h**

где есть макросы:

```
1      ATOMIC_BLOCK(type)
2      {
3        код который будет атомарным;
4      }
```

Где в качестве type возможны два варианта: **ATOMIC_FORCEON** — прерывания по выходу из атомарного блока будут включены и **ATOMIC_RESTORESTATE** в котором состояние флага I будет таким же какое и на входе в блок. Запрещены — так запрещены, разрешены так разрешены. Но работает чуть медленней и памяти требует больше.

Также там есть макрос разатомаривания.

```
1 NONATOMIC_BLOCK(type)
```

```
2       {
3         код который будет неатомарным;
4       }
```

Т.е. он делает обратную операцию — разрешает прерывания в своем чреве. Иногда удобней запретить прерывания во всей функции, но в центре разрешить, чем делать два атомарных блока в начале и в конце.

Синтаксис этих макросов такой же, а опции по аналогии **NONATOMIC_RESTORESTATE** — оставляет как было. **NONATOMIC_FORCEOFF** — принудительно выключает прерывания по выходу.

Так что можно вместо:

```
1 cli();
2 flag_byte |=1<<options;
3 sei();
```

Написать

```
1 ATOMIC_BLOCK(ATOMIC_FORCEON)
2 {
3 flag_byte |=1<<options;
4 }
```

И это будет эквивалентно. Единственное что код потеряет часть переносимости. По хорошему бы вообще вынести все эти cli(); sei(); в отдельный хидер где заменить из на что нибудь вроде Interrupt_Enable(); Interrupt_Disable(); и тогда этот код можно будет перетащить на любой контроллер, лишь бы там был Си компилятор. А уж привязать к Interrupt_Enable(); местный аналог sei(); куда проще в одном файле в одном месте чем перелопачивать весь сырок.

З.Ы.
Также недавно появилась отличная статья Виктора Тимофеева [2] про то как надо правильно писать код. Настоятельно рекомендую к прочтению.

AVR. Учебный Курс. Программирование на Си. Работа с памятью, адреса и указатели

Указатель [1]. Один из самых мутных для понимания и в то же время совершенно необходимый инструмент любого языка программирования. Вызывает массу вопросов и непонимания на начальном этапе обучения.

Итак, начну по порядку.
Инфа, любая инфа (команды, данные) лежит в памяти по ячейкам. У каждой ячейки есть порядковый номер — адрес.

Мы можем напрямую сказать процессору — возьми данные из ячейки с адресом 0xA0 и положи его в ячейку с адресом **0×11**. Это будет прямая адресация. Здесь адреса **0xA0** и **0×11** содержатся напрямую в машинном коде. Это очень быстро, просто и не требует никаких дополнительных телодвижений. Один минус — адреса **0xA0** и **0×11** нельзя изменить, как мы их впишем в код, так они там и останутся.

Но может быть и другой способ. Когда у нас есть еще две ячейки памяти. Например, **А** и **Б** в которые мы предварительно положим числа **0xA0** и **0×11** соответственно. И тогда предыдущая операция будет выглядеть так.

Возьми число из ячейки адрес который лежит в **А** и положи в ячейку адрес которой узнаешь из **Б**.

Результат тот же, но возникло множество дополнительных телодвижений. Во первых положить первоначальные адреса **0xA0** и **0×11** в ячейки **А** и **Б**. Потом, при совершеннии операции, используя данные ячеек **А** и **Б** как адреса, взять уже оттуда нужные нам данные и совершить обмен.

Но прелесть вся в том, что при этом мы можем как угодно менять **А** и **Б** (ведь это такие же переменные как и любые другие) и они будут указывать на разные данные.

А один и тот же кусок кода становится универсальным. Он может работать с любыми данными адреса которых нам укажут переменные **А** и **Б**.

А сами эти переменные и будут указателями.

Вот и вся премудрость.

Чтобы было проще для понимания для ассемблерщиков я буду давать ассемблерную аналогию работы с указателем. Конечно компилятор делает все не так прозрачно — начинает тасовать ячейками памяти и регистрами как заправский фокусник колодой карт. На этом этапе разглядывать ассемблерный листинг сишного кода можно лишь с целью уловить «Общие тенденции в городе». Поэтому я дам не реальный пример из скомпиленного кода, как делал раньше, а упрощенный аналог.

В Си, для работы с указателями есть операнд звездочка (*****). Инициализируется указатель так же и там же где и обычная переменная.
Собственно, указатель переменной и является. Только специфичной — хранящей в себе адрес.

Заведем две переменные **i** и **z**, а еще один указатель **u**

```
1 unsigned char i,z;
2 unsigned char *u;
```

Свежесозданный указатель изначально указывает в никуда. Точнее может быть куда нибудь и показывает, но явно не туда куда надо. Так что его надо нацелить. Для нацеливания на какую либо ячейку памяти или переменную используется амперсанд «**&**«.
Это операнд взятия адреса.

```
1 u=&i;
```

Вот таким простым образом мы взяли и нацелили на переменную **i** указатель **u**.

На ассемблере это будет выглядеть примерно так:
Вначале две переменные в памяти. Назову их по другому, чтобы не путались с сишными.

```
1               .DSEG
2 Mode1:       .byte               1
3 Mode2:       .byte               1
```

Вначале, как водится, грузим адреса переменных в индексные регистры. Собственно регистровые пары X,Y,Z и есть самые настоящие указатели!
Так что:

```
1         LDI     ZL,low(Mode1)   ; Z=&Mode1;
2         LDI     ZH,high(Mode1)  ;Это нацеливание указателя, его инициализация.
```

Теперь мы можем сделать с указателем две вещи.

1) Изменить сам указатель.

Например так:

```
1 u++;
```

При этом указатель перестанет указывать на i и будет показывать на следующую после i ячейку памяти. Это может быть полезно при обработки строк и массивов. Мы указываем на начало строки, а потом, увеличивая указатель, проходимся по всей строке.

На асме:

```
1       SUBI      ZL, Low(-1)     ; Z++;
2       SBCI      ZH, High(-1)    ; Инкремент указателя
```

или

```
1 u=u+j*k;
```

На асме тут придется взять адрес из Z и всяко его математически изменить, а потом сунуть обратно в Z.

Так, например, можно брать данные из массивов.

Указатель можно складывать и вычитать с целочисленной константой ведь с точки зрения компилятора это всего лишь переменная. Нам лишь надо следить чтобы мы не нацелили указатель куда нибудь не туда, к примеру, за границы нашего массива. Иначе все может плохо кончится. Будет хитрая ошибка которую сложно найти.

2) Изменить или пощупать данные на которые указывает указатель.

```
1 (*u)++;    //Инкремент переменной на которую показывает указатель
```

На асме:

```
1                               ; (*Z)++;
2         LD      R17,Z         ; Вначале берем в регистр данные на которые указывает Z
3         INC     R17           ; Инкремент данных на которые указывает указатель.
4         ST      Z,R17         ; А потом сохраняем обратно, оттуда где взяли
```

или так

```
1 *u=m;
```

На асме:

```
1         MOV     R17, m  ; Взяли откуда нибудь нашу m
2         ST      Z,R17   ; и запислаи ее по адресу который в Z
```

При этом надо учитывать приоритет операций. Дело в том, что если мы запишем нашу операцию как

```
1 *u++;
```

На асме:

```
1         SUBI    ZL, Low(-1)     ; Z++;
2         SBCI    ZH, High(-1)    ; Инкремент указателя
3         LD      R17,Z           ; и что дальше?
```

То ничего не с данными произойдет. Операция ++ имеет тот же приоритет что и обращение через указатель * , а выполняются они у нас справа на лево. И у нас произойдет сначала увеличение указателя u, а потом уже мы по новому указателю обратимся в память и ничего там не сделаем. **Подробней про приоритеты операций** [2]

Тип данных указателя
Если указатель это всего лишь адрес, то как же он может иметь тип? В AVR адрес двухбайтный, а на какие данные мы бы не ссылались указателем на длинну адреса это не влияет. Зачем тогда тип указателя?

А затем, чтобы компилятор знал на сколько этот адрес изменять при операциях с указателем.

В ассемблере, ясно дело, типов никаких нету. Поэтому нам надо самим думать на сколько изменять индексные регистры в каждом конкретном случае.

Еще тип позволяет отслеживать правильность обращения к данным через указатель. Например, чтобы мы не пытались записать в байт слово, или вместо структуры загнать всего один байт. Без типа, по одному адресу, мы не сможем понять что нас ждет на том конце указателя. А значит можно наделать кучу багов. А так нас компилятор предупредит Warning'ом. Мол чего это ты, товарищь, пургу гонишь? Так что Warning с указателем можешь смело приравнивать к критической ошибке и искать ее причину. :)

Чтобы соответствовать всем правилам, указатель должен быть того же типа, что и данные на которые он показывает.
То есть на **u16 value;** должен быть **u16 *pointer;**

А если нам надо двубайтные данные разобрать по байтам?
Конечно, можно взять и нацелить на то же двубайтное значение однобайтный указатель. И это будет прекрасно работать.

```
1 u16 value;
2 u08 *pointer_8;
3
4 pointer_8=&value;
```

Но компилятор перекосит от возмущения и он завалит тебя Warning'ами вида: «Pinboard_1.c:43: warning: assignment from incompatible pointer type». Что дескать тип не совпадает и ты часом там не ошибся? В этом случае надо делать преобразование типа указателя. Задав нужный тип явным образом.

```
1 u16 value; // Двубайтная переменная value
2 u08 *pointer_8;  // Указатель на однобайтные данные.
3
4 // Даем явно понять, что несмотря на то, что данные в value двубайтные
5 // Мы будем с ними работать как с байтом
6 pointer_8=(u08 *)&value;
```

Это успокоит компилятор.

Также бывают указатели вообще без типа. С типом void

```
1  void *addres;
```

Это просто адрес. Без типа, без размерности. Тупо два байта указывающие куда то в память. Все.
Для того чтобы с ним сделать какую нибудь гнусную вещь надо сначала решить мальчег это или девАчка. В смысле куда он нацелен и что мы с этим будем делать. Поэтому при использовании указателя типа void мы каждый раз должны явно обозначать тип данных на той стороне провода. Вот так:

```
1 (u16 *)addres++; //Увеличили указатель как будто он u16, т.е. на 2 байта
2
3 (u08 *)addres++; // А теперь словно он u08 -- на один байт
```

Приведу пример.
Есть у нас массив двубайтных слов данных str типа u16 (aka unsigned short). И есть указатель **u** типа u16 который указывает на первый элемент строки.

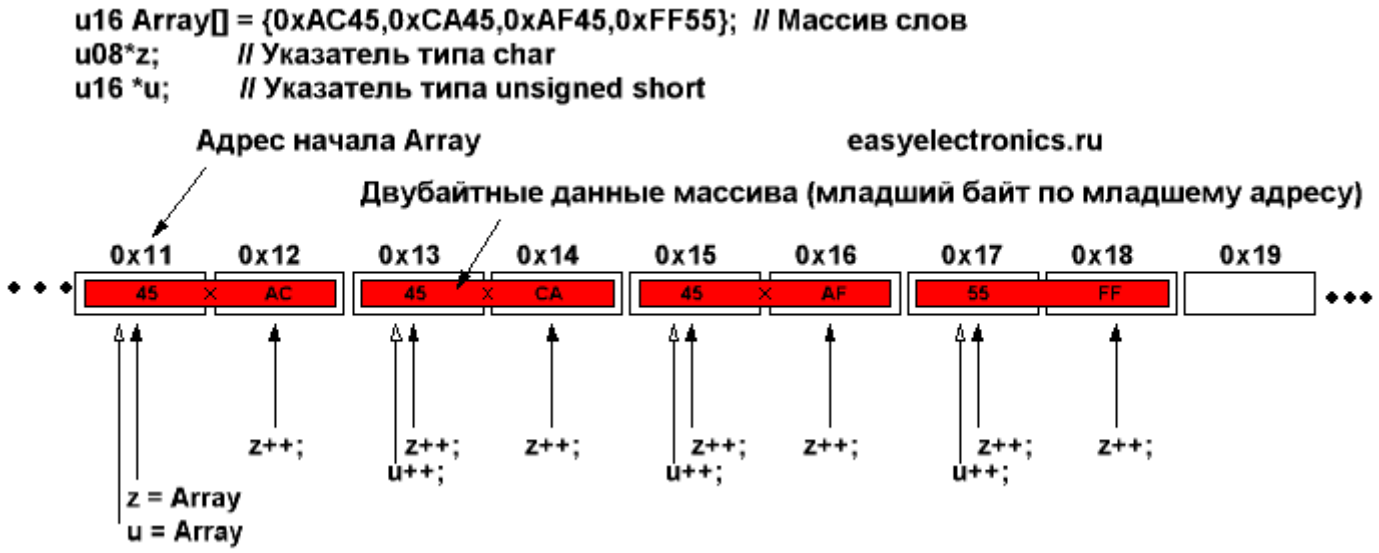
```
1 u16 str[10];
2 u16 *u;
3 u=str;
```

А потом сделали инкремент указателя:

```
1 u++;
```

Куда покажет указатель? Значение его адреса увелчится на два. Т.е. он укажет на следующее слово. Потому что его тип двухбайтный.

А если бы указатель был однобайтного типа, а указывал на двубайтные данные, то инкремент указателя дал бы нам не следующий двубайтный элемент массива, а на второй байт первого элемента. Таким образом, выбирая тип указателя мы можем задавать дискретность его изменения, выбирая те данные что нам нужны.



Дам один простенький пример на работу со строкам через указатель.

```
1  // Задаем всякие переменные. Не стал выносить их в хидер
2  #define F_CPU 8000000L
3  #define XTAL 8000000L
4  #define baudrate 9600L
5  #define bauddivider (XTAL/(16*baudrate)-1)
6  #define HI(x) ((x)>>8)
7  #define LO(x) ((x)& 0xFF)
8
9  // Подключаем библиотеку ввода вывода
10 #include <avr/io.h>
11
12 //Прототипы функций
13 void SendStr(char *string);
14 void SendByte(char byte);
15
16
17 // Поехали!!!
18 int main(void)
19 {
20 char String[] = "Hello Pinboard User!!!";     // Организуем в памяти массив-строку
21 char *u;                                      // И про указатель не забываем.
22
23
24 // Инициализация периферии
25 UBRRL = LO(bauddivider);
26 UBRRH = HI(bauddivider);
27 UCSRA = 0;
28 UCSRB = 1<<RXEN|1<<TXEN|0<<RXCIE|0<<TXCIE;
29 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
30
31 // Главный код
32 u=String;              // Присваиваем указателю адрес начала строки
```

```
33                      // Где оператор взятия адреса "&"? А он в данном случае и не нужен
34                      // Дело в том, что все сложные типы вроде массивов и строк это и
35 есть
36                      // Указатели в чистом виде. Поэтому мы просто присваиваем один
37                      // Указатель к другому. И все. А вот передавай мы один байт или
38 слово
39                      // Пришлось бы брать адрес операндом "&".
40
41 SendStr(u);          // Печатаем строку в USART
42
43 // А тут еще прикольней. Мы инлайново обьявили строку прям в параметре функции)
44 // Так тоже можно. Работает точно также, строка размещается в RAM и в функцию
45 // Передается указатель на ее заголовок.
46 SendStr(" Hello inline String");
47
48 return 0;
49 }
50
51 // Отправка строки
52 void SendStr(char *string)     // А вот как все тут работает
53 {
54 while (*string!='\0')  // Пока первый байт строки не 0 (конец ASCIIZ строки)
55         {
56         SendByte(*string);              // Мы шлем байты из строки
57         string++;               // Не забывая увеличивать указатель,
58         }                       // Выбирая новую букву из строки.
59 }
60
61 // Отправка одного символа
62 void SendByte(char byte)               // Тут тоже элементарно. На входе байт
63 {
64 while(!(UCSRA & (1<<UDRE)));   // Ждем флага готовности USART
   UDR=byte;                    // Засылаем его в USART
   }
```

Крастота? Прелесть?
Ну да, вот только есть тут одно западло. Используя таким образом строки и переменные мы совершенно варварским образом расходуем оперативку. А она у нас тут маленькая, всего килобайт. Причем западло тут двойное. Строки «Hello Pinboard User!!!» и » Hello inline String» вначале вкомпиливаются в флеш, а потом, при старте МК, борзо копируются в ОЗУ и висят там мертвым грузом.

В микроконтроллерах AVR есть несколько видов памяти. Первая это, конечно же, ОЗУ. Оперативка. В ней хранятся все переменные и стек. Доступ к ней осуществляется из Си безо всяких ухищрений. Завел переменную или массив — и вот тебе обращение. То же самое и насчет указателей. Второй тип памяти до которого мы можем дотянуться — флеш. Вот туда бы эти строки и засунуть, точнее, раз уж они там уже есть, брать их сразу оттуда.

Как это сделать?

Тут нам поможет библиотека **pgmspace.h** из стандартной поставки **WinAVR**.
Осталось только определить наши строки с аттрибутом **PROGMEM**

```
1 char String[] PROGMEM = "Hello in FLASH Pinboard User";
```

А для инлайновых строк есть удобный макрос **PSTR**

```
1 SendStr(PSTR(" Hello FLAHS inline String"));
```

Все замечательно, но вот только это работать не будет :(

Дело в том, что память программ адресуется совсем по другому и доступна только через спец команду процессора LPM. Поэтому обычное чтение по указателю тут работать не будет :((впрочем, возможно это косяк именно GCC

т.к. он изначально заточен на Неймановскую архитектуру, а на Гарвардской его перекашивает. Как там в IAR и CVAVR с обращением во флеш через указатели?)

Для этого в **pgmspace.h** есть функции чтения из памяти программ:

```
pgm_read_byte(data);
pgm_read_word(data);
pgm_read_dword(data);
pgm_read_float(data);
```

Как говорится, на любой тип и размер.

Где **data** это передаваемый адрес во флеше.

Поэтому нашу функцию отправки строк придется переделать:

```
void SendStr_P(char *string)   // А вот как все тут работает
{
while (pgm_read_byte(string)!='\0')    // Пока первый байт строки не 0 (конец ASCIIZ строки)
        {
        SendByte(pgm_read_byte(string));       // Мы шлем байты из строки
        string++;                          // Не забывая увеличивать указатель,
        }                                  // Выбирая новую букву из строки.
}
```

Казалось бы все. Теперь работает. Ан нет. Тут у нас есть еще одно западло, стоившее мне когда то часа возни.

Дело в том, что если мы определим нашу **PROGMEM** строку в функции **main**, то компилятор посчитает ее локальной переменной, а все наши ярлыки проигнорирует. Выдаст Warning, а строка будет по прежнему предательски скопирована в RAM. Разумеется через pgm_read_*** до нее уже будет не достучаться. И вся наша программа полетит в тартар.

Чтобы такого не случилось надо все **PROGMEM** константы размещать за пределами **main**. Разумеется это не касается макроса инлайновой вставки **PSTR**, тут можно не париться — он сам все разместит где надо.

Да, кстати, когда мы обьявляем строку (или массив), то ее имя по факту и есть указатель. Так что вполне работает и такой механизм:

```
SendStr_P(StringP);
```

Ну а вся прога, со всеми заморочками, стала такого вида:

```
#define F_CPU 8000000L
#define XTAL 8000000L
#define baudrate 9600L
#define bauddivider (XTAL/(16*baudrate)-1)
#define HI(x) ((x)>>8)
#define LO(x) ((x)& 0xFF)


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

//Прототипы функций
void SendStr(char *string);
void SendStr_P(char *string);
void SendByte(char byte);
//_____________________________

```

```
18 char StringP[] PROGMEM = " Hello_IN_FLASH";
19
20 int main(void)
21 {
22 char String[] = " Hello_IN_RAM";
23
24 char *u,*z;
25
26 // Инициализация периферии
27 UBRRL = LO(bauddivider);
28 UBRRH = HI(bauddivider);
29 UCSRA = 0;
30 UCSRB = 1<<RXEN|1<<TXEN|0<<RXCIE|0<<TXCIE;
31 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
32
33
34 // Главный код
35 u=String;               //Копируем указатель на RAM
36
37 z=StringP;              // И он ничем не отличается от указателя на FLASH
38                         // И в том и в другом случае это двубайтный адрес.
39 // Так что работа с ним одинаковая. Разница лишь в том Из какой памяти черпать эти
40 // данные? Для этого и нужны все эти pgm_read_***.
41
42 SendStr(u);                             //Печатаем данные по первому указателю
43 SendStr(" Hello_INLINE_IN_RAM");       // Печатаем инлайную строку. Она копируется и в
44 RAM
45                                        // и во FLASH!!! Не оптимально!!!
46
47 SendStr_P(z);                                    // Печатаем по указателю из флеша
48 SendStr_P(PSTR("Hello_INLINE_IN_FLASH"));      // Инлайновая флеш строка
49 SendStr_P(StringP);                             // Печатаем по прямому адресу строки.
50                                                 // Он ведь тоже такой же указатель как и u
51
52
53 return 0;              // Конец программы.
54 }
55
56 // Отправка строки
57 void SendStr(char *string)
58 {
59 while (*string!='\0')
60         {
61         SendByte(*string);
62         string++;
63         }
64 }
65
66 // Отправка строки из флеша
67 void SendStr_P(char *string)
68 {
69 while (pgm_read_byte(string)!='\0')
70         {
71         SendByte(pgm_read_byte(string));
72         string++;
73         }
74 }
75
76 // Отправка одного символа
77 void SendByte(char byte)
78 {
79 while(!(UCSRA & (1<<UDRE)));
80 UDR=byte;
```

```
}
```

А если нам надо разместить в памяти здоровенный массив строк? Как быть? Обычный прикол вида:

```
char *string_table[]  =
{
    "String 1",
    "String 2",
    "String 3",
    "String 4",
    "String 5"
};
```

Для PROGMEM не прокатит. Тут надо делать в памяти отдельные стркои и увязывать их массивом укзателей.

```
char MenuItem0[] PROGMEM = "Menu Item 0";  // Это наши строки
char MenuItem1[] PROGMEM = "Menu Item 1";
char MenuItem2[] PROGMEM = "Menu Item 2";

// А это наш массив указателей. Обрати внимание, что тип такой же как у строк.
char *MenuItemPointers[] PROGMEM = {MenuItem0, MenuItem1, MenuItem2};
```

Строки самые обычные. Их можно напечатать нашей функцией **SendStr_P** если скормить ей указатель(хотя бы имя строки, как мы это делали выше).

```
SendStr_P(MenuItem0);
```

И строка «**Menu Item 0**» улетит в **UART**. Но нам то надо совсем другое! Нам надо брать строки по индексам из списка **MenuItemPointers**.

Казалось бы, в чем проблема то? Берем да запихиваем в нашу функцию **SendStr_P** элемент массива **MenuItemPointers[1]**.

```
SendStr_P(MenuItemPointers[1]);
```

Компилятор эту матрешку развернет и достанет из нее указатель на искомую строку и зашлет все в USART. Да не тут то было! Проблема опять в том, что **MenuItemPointers** опять лежит во флеше и так просто до ее элементов не добраться. Только через **pgm_read_*****.

В результате получаем такую вот загогулину:

```
SendStr_P((char*)pgm_read_word(&(MenuItemPointers[1])));
```

А на самом деле все просто:

- Массив указателей лежит во флеше. Указатель это двубайтный адрес — **word**. Чтобы его достать оттуда нам нужна **pgm_read_word** которой мы скармливаем адрес ячейки **MenuItemPointers[1]**.
- Сам заголовок массива **MenuItemPointers** это указатель в чистом виде, поэтому ему бы **&** не потребовался. Но вот конкретный элемент массива (в частности [1], хотя может быть и [i]) это уже переменная и на нее нужно узнавать адрес через **&**. Поэтому и делаем вычисление адреса
- **pgm_read_word** нам достает из флеша **word** который уже является адресом **MenuItem1** в чистом виде. И можно было бы так и оставить, скормив его нашей функции **SendStr_P**. Она все равно ничего другого не ест. Но вот только компилятор тут дико взвоет Warning!!! ЧТО ЭТО ТЫ ТАМ ИЗ БЕЗДНЫ ДОСТАЛ??? ЙА БОЙУСЯЯЯЯ!!! А ты ему — «Не сцы, это наш старина однойбайтный указатель **(char*)**» и делаешь явное указание типа.

А целиком код выглядит так:

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

//Прототипы функций
void SendStr(char *string);
void SendStr_P(char *string);
void SendByte(char byte);
//_____________________________


// Строки флеша размещать ЗА пределами main!!!
char MenuItem0[] PROGMEM = "Menu Item 0";
char MenuItem1[] PROGMEM = "Menu Item 1";
char MenuItem2[] PROGMEM = "Menu Item 2";
char *MenuItemPointers[] PROGMEM = {MenuItem0, MenuItem1, MenuItem2};

int main(void)
{
// Инициализация периферии
UBRRL = LO(bauddivider);
UBRRH = HI(bauddivider);
UCSRA = 0;
UCSRB = 1<<RXEN|1<<TXEN|0<<RXCIE|0<<TXCIE;
UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;


// Главный код
SendStr_P(MenuItem0);  // Вывод строк по прямому адресу строки

SendStr_P((char*)pgm_read_word(&(MenuItemPointers[1]))); // Вывод по таблице
SendStr_P((char*)pgm_read_word(&(MenuItemPointers[2])));

return 0;
}

// Отправка строки
void SendStr(char *string)
{
while (*string!='\0')
        {
        SendByte(*string);
        string++;
        }
}

// Отправка строки из флеша
void SendStr_P(char *string)
{
while (pgm_read_byte(string)!='\0')
        {
        SendByte(pgm_read_byte(string));
        string++;
        }
}

// Отправка одного символа
void SendByte(char byte)
{
while(!(UCSRA & (1<<UDRE)));
UDR=byte;
}
```

Ну как? Постиг Дзен Си? Казалось бы, все просто. Все логично. Однако, не зная с чего начать тут можн очень долго ходить кругами. А как хорошо на ассемблере — у нас есть указатель Z и забойная команда LPM. А дальше Ать-ать-ать и сами накруичиваем любые матрешки из вложенных подропрограмм, потрошащих флеш вдоль, поперек и по диагонали. Благодать!

Указатель может быть не только на данные, но и на функцию. Ведь что такое функция? Это всего лишь кусок кода с адресом во флеше. Не более того! А все вызовы функций, все эти myfunc(); это указание компилятору скакнуть через CALL/ICALL на адрес myfunc. Так что нам мешает сделать на этот адрес указатель, а потом по нему перейти? Правильно — ничего. Делаем:

```
1  void func1(void)
2  {;}
3  void func2(void)
4  {;}
5
6  int main(void)
7  {
8  void (*f)(void);        // указатель на функцию
9  f = func1;
10 f();             //выполнит функции func1()
11 f = func2;
12 f();             //теперь выполнит функцию func2()
13 }
```

Зачем такой извраг? Ну много применений можно придумать. Например, байт код виртуальной машины. Где у тебя есть скрипт в виде массива действий, где каждое действие это адрес на функцию это действо выполняющее. А твоя программа, виртуальная машина, хватает из массива адреса и переходит на функции. А те, в свою очередь, делают что то, меняют порядок действий (если нужно). В общем, программа в программе получается :)
Или, второй пример, очередь задач в диспетчере ОС. Где у нас не прямые вывозовы функций, а заброс их в конвеер, где они выполняются в порядке очереди. Об этом я еще расскажу после.

AVR toolchain своими руками

DI HALT:
В догонку к прошлому посту про AVR Studio в Linux [1] досылаю и про сборку avr-libc там же. Вынесено из комментов к предыдущему посту. Спасибо Dark SavanT

Если есть возможность поставить готовый тулчейн из пакета, лучше воспользоваться ей.
преимущество самосборного в том, что все что надо, лежит там где сказано и не засоряет /usr/*. Но тут мы теряем автоматические обновления из пакетов. Короче, думайте сами, решайте сами.

Поехали!
В минимальной комплектации нам понадобятся:

- **binutils** — это ассемблер, линкер, objdump и еще куча необходимых для сборки бинарника вещей
- **gcc** — собственно сам компилятор.
- **avr-libc** — стандартная библиотека C для AVR архитектуры.
- **avrdude** — программа для прошивки. Хальт про нее не раз писал.

для того, чтобы это все безобразие собралось, нужен установленный в системе gcc, bash, awk, binutils, libc, может что-то еще забыл.

делаем каталог для сборки:

```
mkdir build
cd build
```

качаем исходники

```
wget -c http://ftp.gnu.org/gnu/binutils/binutils-2.20.tar.gz
wget -c http://ftp.gnu.org/gnu/gcc/gcc-4.4.2/gcc-4.4.2.tar.bz2
wget -c http://mirror.vocabbuilder.net/savannah/avr-libc/avr-libc-1.6.7.tar.bz2
```

поддиректории для сборки отдельных компонентов

```
mkdir build/binutils
mkdir build/gcc
mkdir build/libc
```

распаковываем

```
tar zxvf binutils-2.20.tar.gz
tar jxvf avr-libc-1.6.7.tar.bz2
tar jxvf gcc-4.4.2.tar.bz2
```

устанавливаем переменные окружения:
PREFIX — для того, чтобы **make install** скопировал бинарники в нужную директорию
PATH — для того, чтобы собрать libc компилятором для avr

```
export PREFIX=/opt/avrchain
export PATH=$PATH:$PREFIX/bin
```

Собираем **binutils**

```
cd build/binutils
../../configure --prefix=$PREFIX --target=avr --disable-nls
make install
```

Собираем **gcc**

```
cd ../gcc
../../gcc-4.4.2/configure --prefix=$PREFIX --target=avr --enable-languages=c --disable-nls
--disable-libssp --with-dwarf2
make install
```

Собираем **avr-libc**

```
cd ../libc

../avr-libc-1.6.7/configure --prefix=$PREFIX --host=avr

make install
```

Проверим

```
[savant@savant-laptop avrchain]$ avr-gcc --version
avr-gcc (GCC) 4.4.2
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

должны получить такой вывод.

А теперь что-нибудь соберем.
Я не запаривался и взял простой пример из AVR Курса [2]

Вариант №1: все руками

Компилим:

```
avr-gcc -g -mmcu=atmega8 Pinboard_1.c -o demo.o
```

Линкуем:

```
avr-gcc -g -mmcu=atmega8 -o demo.out demo.o
```

Получаем .hex фаил для заливки в МК

```
avr-objcopy -j .text -O ihex ./demo.out demo.hex
```

Вариант №2: Makefile
создаем рядом с исходником Makefile такого вида:

```
CC=avr-gcc
OBJCOPY=avr-objcopy

CFLAGS=-g -mmcu=atmega8

all : demo.out
        $(OBJCOPY) -j .text -O ihex demo.out rom.hex

demo.out : demo.o
        $(CC) $(CFLAGS) -o demo.out -Wl,-Map,demo.map demo.o

demo.o :
        $(CC) $(CFLAGS) -Os -c Pinboard_1.c -o demo.o
```

далее говорим **make** и в итоге получаем hex фаил для заливки в контроллер. Как дальше жить, сами знаете, чай не маленькие, avrdude пользовать умеете.

AVR. Учебный Курс. Архитектура Программ

Все учебные курсы по микроконтроллерам которые я встречал (в том числе, к сожалению, и мой ассемблерный, но я надеюсь это постепенно поправить) страдают одной и той же проблемой.
В курсе бросаются строить дом не заложив фундамент. Только показав на примере как мигнуть светодиодом, сразу же кидаются в периферию. Начинают осваивать ШИМ-ы, таймеры, подключать дисплеи и всякие термодатчики.
С одной стороны это понятно — хочется действа и результата мгновенно. С другой — рано или поздно такой подход упрется в тот факт, что программа, надстраиваемая без четкой идеологии, просто обрушится под своей сложностью. Сделав невозможным дальнейшее развитие.
Итогом становится либо рождение жутких программных уродцев, либо миллионы вопросов на форуме вида «а как бы мне все сделать одновременно, а то частоты контроллера уже на все не хватает».

Самое интересное, что правильной организации программы учат программистов в ВУЗах, но вот только к микроконтроллеру народ обычно идет не от программирования, а от железа. А, как показала практика обучения в ВУЗе, электронщиков толковому программингу практически не обучают :(Приходится все додумывать самостоятельно.

Итак, что такое структура программы. Это, прежде всего, ее скелет. То какими путями движется код. Как организованы переходы между задачами прошивки. То как распределяется процессорное время. Без краткого ликбеза по общим принципам построения прошивки дальше двигаться нет смысла.

Все ниже написанное это лишь продукт моих умозаключений, поэтому терминология может отличаться от общепринятой. Если это сильно кому то будет резать глаз — поправляйте в комментах.

Итак, я для себя выделяю следующие структуры, по порядку возростания сложности конструкции и количеству управляющего кода:

- Суперцикл

- Суперцикл+прерывания
- Флаговый автомат
- Диспетчер
- Приоритетный диспетчер
- Кооперативная RTOS
- Вытесняющая RTOS

А теперь подробно по каждому пункту:

Суперцикл
Самая простая организация программы. Отличается минимальным количеством управляющего кода (код который не выполняет полезную работу, а служит лишь для организации правильного порядка действий). Идеально подходит для задач из серии «помигать диодом».

Алгоритм прост как мычание (псевдо код):

```
1  void main(void);
2  {
3  while(1)
4          {
5          Led_ON();
6          Delay(1000);
7          Led_OFF();
8
9          u=KeyScan();
10         switch(u)
11                 {
12                 case 1: Action1();
13                 case 2: Action2();
14                 case 3: Action3();
15                 }
16         }
17 }
```

И все примерно в таком духе. Т.е. тупое последовательное выполнение кода. Задержка — прям там же, в общей куче. Опрос клавиатуры. Тут же. Внутри суперцикла могут быть переходы и ветвления, но суть остается та же самая — тупое движение по коду.
Достоинства очевидны сразу логичность и простота (поначалу, потом это ад). Недостатки тоже вылазят сразу же — чем больше мы запихнем в наш код, тем он будет более неповоротливым и тормозным.
Однако суперцикл поддается оптимизации. Например, в функцию Delay() можно не тупо впустую щелкать тактами, а запихать туда опрос клавиатуры и еще кучу полезного экшна.
В итоге, на суперцикле можно сделать сверх компактную, надежную, но при этом совершенно монолитную программу. В нее будет нельзя ни добавить ни отнять. А еще чтобы все учесть, написать и отладить надо быть гением. Читали историю про Один байт? [1] Вот это наверняка про нее или про следующий вариант.
Обычно народ, наигравшись с суперциклом, раскуривает прерывания и быстро переходит к варианту суперцикл+прерывания.

Суперцикл+прерывания
Чистый суперцикл используется крайне редко, потому как в любом микроконтроллере куча периферии и у них есть такая удобная вещь как прерывания (Хотя, я встречал на форуме microchip.ru высказывание что прерывания снижают надежность программы и лучше избавляться от них по возможности) .

Здесь ситуация становится несколько иной. За счет прерываний у нас появляются параллельные процессы. Например, мы можем повесить опрос клавиатуры и мигание лампочкой на прерывание по таймеру (псевдокод):

```
1  ISR(Timer1)
2  {
3  u=KeyScan();
4  }
5
6  ISR(Timer2)
7  {
```

```
8  if(LED_ON)
9          {
10         Led_OFF();
11         }
12         else
13         {
14         Led_ON();
15         }
16 }
17
18 void main(void);
19 {
20 Timer1_Delay=100;       // Выдержка таймера1 100
21 Timer1=1<<ON;           // Включить таймер1
22
23 Timer2_Delay=1000;      // Выдержка таймера2 1000
24 Timer2=1<<ON;           // Включить таймер2
25
26 SEI();                  // Разрешить прерывания
27 while(1)
28         {
29         switch(u)
30                 {
31                 case 1: Action1();
32                 case 2: Action2();
33                 case 3: Action3();
34                 }
35         }
36 }
```

Уже лучше. Вроде как программа теперь крутится тремя независимыми процессами. Нигде не тормозит и все в порядке. Да, до тех пор пока хватает аппаратных ресурсов. У нас ведь что тут получается — на каждый процесс по прерыванию. А таймеров то всего ничего и прерывания грузить тяжелыми по времени процессами нельзя — они должны быть быстрыми.

И вот именно в этом месте сон разума начинает рождать чудовищ. Появляются какие то дополнительные условия, проверки. На одно прерывание вешается куча разных событий, проверка флажков, попытка все это дело увязать… В результате конструкция превращается в макаронный клубок из переходов, вызовов, условий… В котором что либо изменить или понять невозможно. Не имея фундамента, конструкция рушится под своим весом. Разрушая мозг своему создателю. Начинается анархия…

Что делать? А что делать когда возникает анархия? Правильно! Нужно какое то единое правление — бюрократический аппарат. С теоретической точки зрения от него толку как от козла молока — клавиатуры он не опрашивает, ножками не дрыгает, данные не шлет. Только место занимает и процессорное время жрет. Но без него как без рук.

Флаговый автомат
Первый управляющий механизм это классический флаговый автомат. Реализаций его существует миллион, каждый придумывает свой собственный вариант, но разницы между ними большой нет.

Итак, в суперциклах у нас задачи вызывались одна за другой. Либо в порядке общей очереди, либо в виде прямого вызова. Очевидно, что эта цепь получается неразрывной и если в ней есть затыки (например какой нибудь DELAY), то встает вся цепь.
Флаговый автомат позволяет разорвать эту цепь.

Вот его простейший вариант.
Сначала мы определяем флаги. Обычно это битовое поле [2], где в одном байте насовано несколько значащих битов. Каждый за что нибудь да отвечает. Особые извращенцы под каждый флаг заюзывают по байту.

Пусть у нас будет байт flag, а в нем разные биты, выделяемые на нашем псевдокоде точкой. Итак, выглядит примерно это так (псевдокод):

```
void main(void)
{

//Start Task
flag.ScanKey=1; //Запустить сканирование клавиатуры
flag.Led_On=1;  // Запустить мигалку

while(1)
        {
        if(flag.keyscan==1)
                {
                u=ScanKey();
                switch(u)
                        {
                        case 1: flag.Action1=1;        // Поставить задачу1 на выполнение
                        case 2: flag.Action2=1;        // Поставить задачу2 на выполнение
                        case 3: flag.Action3=1;        // Поставить задачу2 на выполнение
                        }
                }

        if(flag.led_On==1) Led_ON();   // Если стоит флаг включения - включить
        if(flag.Led_Off==1) Led_OFF(); // Если стоит флаг выключения - выключить

        if(flag.Action1==1) Action1(); // Если надо выполнить задачу 1 - выполнить
        if(flag.Action2==1) Action2(); // Если надо выполнить задачу 2 - выполнить
        if(flag.Action3==1) Action3(); // Если надо выполнить задачу 3 - выполнить
        }
}
```

Задача может выглядеть примерно так (всевдокод):

```
void Action1(void)
{
flag.Action1=0;                        // Задача выполнена. Сбросить флаг
DoSomeThing();                  // Делаем что то полезное
if(Some_Event) flag.Action3=1; // Если какое то условие выполнено, то поставим на
выполнение
                                // Задачу 3
}
```

Как видишь, тут нет прямой передачи управления между блоками. Все делается через флаги. Надо запустить задачу мы не передаем ей управление, а ставим флаг ее запуска. И при следующей итерации главного цикла задача будет выполнена, а флаг сброшен (или не сброшен, если задача запущена на циклическое исполнение). Наращивание функционала идет без особых усилий — мы просто добавляем новые флаги и новые секции **if(flag.****==1) { }**

Только у нас лампочка должна была там мигать. Что делать с разнокалиберными временными задержками? Ведь флаговый автомат эту проблему так и не решил. Да, сам по себе флаговый автомат бесполезен. До тех пор пока на арену не вылазит…

Программный таймер
Могучее средство работы с временными задержками. Позволяет с помощью всего одного аппаратного таймера легко держать под контролем прорву задержек разной длительности.

Для начала надо сконфигурировать системный таймер, чтобы он генерировал нам системные тики. У меня за один тик принята 1мс. Поэтому я настраиваю таймер таким образом, чтобы он мне каждую миллисекунду генерировал прерывание. А вот в его прерывании мы и учиним беспредел.

Программный таймер обычно делается в виде массива структур (псевдокод):

```
volatile static struct                                  // Глобальная переменная
```

```
2                   {
3                   Number;                                 // Номер флага в флаговом байте
4                   Time;                                       // Выдержка в мс
5                   }
6                   SoftTimer[Max_Numbers_of_Timer];    // Очередь таймеров
```

А в прерывании по таймеру гоним примерно следующий код (псевдокод):

```
   ISR(Timer1)
   {
1          for(i=0;i!=Max_Numbers_of_Timer;i++)       // Прочесываем очередь таймеров
2          {
3           if(SoftTimer[i].Number == 255 ) continue; // Если нашли пустышку - следующая
4   итерация
5
6           if(SoftTimer[i].Time !=0)              // Если таймер не выщелкал, то щелкаем еще
7   раз.
8                   {
9                   SoftTimer[i].Time --;    // Уменьшаем число в ячейке если не конец.
10                  }
11                  else
12                  {
13                  flags |= 1<<Number ;     // Дощелкали до нуля? Взводим флаг в флаговом
14  байте
15                  SoftTimer[i].Number = 255;     // А в ячейку пишем затычку -- таймер пуст.
16                  }
17         }
   }
```

Видишь, все просто. Мы прочесываем массив структур таймеров, поэлементно. Если в поле **Number** у нас 255, то очевидно что это пустой таймер. Т.к. номер флага таким быть не может (в качестве номера флага может быть только число с одним единичным битом). Такой таймерный слот пропускается.
Если таймер не пуст, то мы проверяем поле Time на ноль. Если не ноль, то уменьшаем и переходим к следующему элементу массива.
А коли таймер дощелкал, то мы устанавливаем взводим во флаговом регистре бит лежащий в поле **Number**. И при следующем прогоне главного цикла управляющая конструкция **if(flag.***==1)** Запустит нам нужную задачу.

Число таймеров определяется исходя из числа одновременно отсчитываемых временных интервалов. Мне обычно хватает десятка. Если сделать меньше — может не хватит и получишь срыв таймерной очереди. Если сделать с большим запасом, то очередь будет дольше обрабатываться в прерывании, а это снижает точность отсчетов интервалов, да и вообще возникает лишний затык.

Ставится таймер откуда угодно функцией **SetTimer** примерно такого вида (псевдокод):

```
1  void SetTimer(NewNumber,NewTime)
2  {
3  InterruptDisable();                    // Запрещаем прерывания. Помним об атомарном
4  доступе!
5
6  for(i=0;i!=Max_Numbers_of_Timer;i++)          //Прочесываем очередь таймеров. Ищем нет ли
7  уже там такого
8          {
9          if(SoftTimer[i].Number == NewNumber)   // Если уже есть запись с таким флагом
10                 {
11                 SoftTimer[i].Time = NewTime;   // Перезаписываем ей выдержку
12                 InterruptRestore();
13                 return;                        // Выходим.
14                 }
15         }
16
17 for(i=0;i!=Max_Numbers_of_Timer;i++)   //Если не находим, то ищем любой пустой
18         {
```

```
19          if (SoftTimer[i].Number == 255)
20                  {
21                  SoftTimer[i].Number = NewNumber;        // Заполняем поле флага
22                  SoftTimer[i].Time = NewTime;            // И поле выдержки времени
23                  InterruptRestore();
24                  return;                                 // Выход.
25                  }
26          }
27 InterruptRestore();     // Восстанавливаем прерывания как было.
    // тут можно сделать return с кодом ошибки - нет свободных таймеров
   }
```

Работает тоже не сложно.
На входе у нас два значения — время **NewTime** и флаг который мы должны воткнуть.
Мы прочесываем нашу очередь таймеров, в поисках свободной ячейки либо таймера на то же событие с еще не истекшим сроком. Если находим такой же таймер — апдейтим его на новое время. Если не находим, то втыкаем данные впервую свободную ячейку. А если не нашли и свободной ячейки, то получаем Timer Fail. Это ошибка, ее надо учитывать делая очередь с запасом, либо точно высчитывая сколько таймеров нам надо.

Теперь, вооруженные знанием о софтверных таймерах, рассмотрим как будет организована наша мигалка (псевдокод):

```
1  void LED_ON(void)
2  {
3  flag.led_On=0;                  // Отработали, флаг можно сбросить
4  SET_BIT_LED();                  // Собственно, зажгли что то там
5  SetTimer(OFF_LED_FLAG,1000);    // Поставить флаг на погашение через 1с
6  }
7
8  void LED_OFF(void)
9  {
10 flag.led_Off=0;          // Отработали, флаг можно сбросить
11 CLR_BIT_LED();                  // Собственно, погасили что то там
12 SetTimer(ON_LED_FLAG,1000);     // Поставить флаг на зажжение через 1с
13 }
```

Все! Видите какие простые и линейные получаются кусочки. И насовать в эту систему еще с полтора десятка цепочек не составит труда. Добавляем флаги и условия, происываем таймеры и вперед. Причем флаги мы можем ставить как в задачах, так и в прерываниях. А главный цикл будет с хрустом это пережевывать.

Думаю понятно, что в такой организации скорость работы главного цикла является критичной. В том плане что всякие хардверные тупые задержки вроде **delay_ms(****)** в ней КРАЙНЕ НЕЖЕЛАТЕЛЬНЫ. Т.к. они тормозят весь конвейер. Все должно быть сделано через службу таймеров. Если нужны очень короткие задержки, то можно завести второй аппаратный таймер с такой же байдой, но тикающий чаще. Но тут надо думать и смотреть.

Еще одним недостатком такой организации является жесткий порядок выполнения операций. Т.е. пока мы не пробежим по всей цепочке **if(flag.***==1)** мы не сможем выполнить какую либо операцию заново (хотя если цепь быстрая, то редко когда критично).
Это частично решается переходом на конструкцию на базе SWITCH-CASE конструкции с выходом в корень после каждой операции. В этом случае у нас возникает другое западло — часто вызываемые задачи могут заблокировать выполнение более медленных.

Думаю принцип понятен, поэтому я не буду приводить работающий пример кода. Мне просто лень :) Если кто то пойдет таким путем то я с радостью выложу его проект в статью. Дело в том, что я предпочитаю другую организацию — динамический диспетчер. Или просто диспетчер. И вот для нее я и приведу пример реального кода. На нем же будут дальнейшие примеры курса :)

Но об этом в следующем посте, а то Война и Мир получается. Оставайтесь на линии!

AVR. Учебный Курс. Архитектура Программ Часть 2

Диспетчер
Данная организация программы требует чуть большего количества кода чем флаговый автомат (Хотя это еще как посмотреть. С увеличением числа задач служебный код динамического диспетчра не увеличивается, а вот флаговый автомат разрастается за счет большего числа флагов и проверок этих флагов, плюс быстродейтсвие снижается, чего нет в диспетчере) , но зато лишена ряда недостатков.

Во первых тут очередь выполнения задач не жестко заданная, а динамическая, конвеерного типа. То есть у нас есть в памяти массив из указателей на задачи-функции. Диспетчер берет указатель и, если он не указывает на Idle, осуществлет переход по этому адресу. Предварительно удалив его из очереди и подкинув очередь.

Заброс указателей-задач в очередь осуществляется другими задачами и прерываниями, а также программными таймерами. Собственно, принцип передачи управления от задачи к задаче похож на флаговый автомат — работаем через посредника. Только у нас тут отстутствуют проверки флагов, а переход делается диспетчером. За счет этого увеличение числа задач не сказывается на увеличении размера управляющей структуры.

Благодаря такой системе управляющую структуру можно вынести в отдельную библиотеку, сварганить к ней конфиг и таскать за собой ее туда сюда. Очень удобно.

А теперь подробно распишу тот диспетчер который стоит в 90% моих проектов на Си.

Очередь задач
Основа основ. С нее все начинается.

Первым делом определяем тип TPTR — Task Pointer. Это обычный указатель пустышка. Просто адрес, без типа.

```
1 typedef void (*TPTR)(void);
```

Потом рисуем очередь задач. Она у нас как глобальная переменная, защищенная от посягательств оптимизатора.

```
1 volatile static TPTR   TaskQueue[TaskQueueSize+1];          // очередь указателей
```

TaskQueueSize это предельное число задач в очереди. Надо делать с некоторым запасом. Задается в дефайнах конфига диспетчера.

Диспетчер задач

```
1  inline void TaskManager(void)
2  {
3  u08      index=0;
4  TPTR   GoToTask = Idle;       // Инициализируем переменные
5
6  // Как видишь, тут есть указатель Idle - ведущий на процедуру простоя ядра.
7  // На нее можно повесить что нибудь совсем фоновое, например отладочные примочки =)
8  // И локальная переменная-указатель GoToTask куда мы будем жрать адреса переходов
9
10 Disable_Interrupt
11 // Запрещаем прерывания!!! Это макрос. Поэтому без ; в конце.
12 // Почему не CLI()? Это команда AVR, а я хотел сделать максимально
13 // платформонезависимый диспетчер. Прерывания надо запрещать потому, что
14 // Идет обращение к глобальной очереди диспетчера. Ее могут менять и прерывания
15 // Поэтому заботимся об атомарности операции.
16
17 GoToTask = TaskQueue[0];       // Хватаем первое значение из очереди
18
19 if (GoToTask==Idle)            // Если там пусто
20         {
21         Enable_Interrupt              // Разрешаем прерывания
22         (Idle)();                     // Переходим на обработку пустого цикла
```

```
23          }
24 else
25          {
26          for(index=0;index!=TaskQueueSize;index++)     // В противном случае сдвигаем всю
27 очередь
28                  {
29                  TaskQueue[index]=TaskQueue[index+1];
30                  }       TaskQueue[TaskQueueSize]= Idle;       // В последнюю запись
31 пихаем затычку Idle
32
33          Enable_Interrupt                // Разрешаем прерывания
34          (GoToTask)();           // Переходим к задаче
            }
    }
```

Постановщик в очередь

```
    void SetTask(TPTR TS)
    {
1   u08       index = 0;
2   u08       nointerrupted = 0;
3
4   if (STATUS_REG & (1<<Interrupt_Flag))  // Если прерывания разрешены, то запрещаем их.
5           {
6           Disable_Interrupt
7           nointerrupted = 1;      // И ставим флаг, что мы не в прерывании.
8           }
9   // Это бодояга вида ATOMIC RESTORSTATE только собственной выделки с закосом под
10  // мультиплатформенность. Как видишь, тут SREG явно не указывается, он прописан в
11  // дефайнах. При переносе на другой микроконтроллер, например, на C51 мне только
12  // пару файлов поправить. А прерывания надо однозначно запретить. Ибо нужно
13  // обеспечить атомарность операций.
14
15
16  // А вот и постановка задачи в очередь.
17  while(TaskQueue[index]!=Idle)  // Прочесываем очередь задач на предмет свободной ячейки
18          {                       // с значением Idle - конец очереди.
19          index++;
20          if (index==TaskQueueSize+1)    // Если очередь переполнена то выходим не солоно
21  хлебавши
22                  {
23                  if (nointerrupted)   Enable_Interrupt // Если мы не в прерывании, то
24  разрешаем прерывания
25                  return;         // Раньше функция возвращала код ошибки - очередь
26  переполнена. Пока убрал.
27                  }
28          }
29  // Если нашли свободное место, то
30  TaskQueue[index] = TS;          // Записываем в очередь задачу
31  if (nointerrupted) Enable_Interrupt    // И включаем прерывания если не в обработчике
    прерывания.
    }
```

Используется просто — если хотим вызывать другую задачу, то мы не ставим флаг, как в флаговом автомате, а пихаем ее адрес в очередь.

```
1 SetTask(Task1);
```

А сама задача выглядит как обычная функция:

```
1 void Task1(void)
```

```
2 {
3 DoSomething();
4 }
```

Причем, естественно можно из задачи делать вызов функций напрямую. Но не стоит надолго затягивать выполнение задачи. Тут как с прерываниями — лучше как можно быстрей передать управление диспетчеру. И по возможности вызовы делать через него. Да, это потребует большей длинны конвеера, но зато код будет работать равномерней.

Очередь таймеров
Разумеется тут нужен и софтверный таймер. Без которого жизнь уже не мила. Начинается он естстественно с очереди таймеров. Которая сделана в виде массива структур. Глобальная переменная, также защищенная от посягательств оптимизатора.

```
1 volatile static struct
2                   {
3                   TPTR GoToTask;              // Указатель перехода
4                   u16 Time;                   // Выдержка в мс
5                   }
6                   MainTimer[MainTimerQueueSize+1];    // Очередь таймеров
```

Состоит из двубайтного счетчика времени и указателя TPTR.

Служба таймера
В прерывании таймера, вызываемого каждую 1мс выполняетя функция обработчика таймера. Обьявлена как инлайновая, что встраивает ее прям в обработчик прерывания, без лишних переходов. Экономим стек, ага

```
   inline void TimerService(void)
1  {
2  u08 index;
3
4  for(index=0;index!=MainTimerQueueSize+1;index++)       // Прочесываем очередь таймеров
5          {
6          if(MainTimer[index].GoToTask == Idle) continue;  // Если нашли пустышку - щелкаем
   следующую итерацию
7
8          if(MainTimer[index].Time !=1)  // Если таймер не выщелкал, то щелкаем еще раз.
9                  {                      // To Do: Вычислить по тактам, что лучш;е !=1 или
10 !=0.
11                 MainTimer[index].Time --;      // Уменьшаем число в ячейке если не конец.
12                 }
13                 else
14                 {
15                 SetTask(MainTimer[index].GoToTask); // Дощелкали до нуля? Пихаем в
16 очередь задачу
17                 MainTimer[index].GoToTask = Idle;   // А в ячейку пишем затычку
18                 }
19         }
   }
```

Прерывание таймера
Служба таймеров пихается в обработчик прерывания от таймера. Каким образом будет делаться прерывание это уже частности. Я сделал на ШИМ таймере по достижении сравнения. Впрочем, можно было повесить и на самый глупый таймер, например на таймер0, который только тикать и умеет. А большего нам и не надо. Но тут придется перезагружать его значение в каждом заходе, чтобы поддерживать постоянное время. А на ШИМе это автоматом идет. Разумеется вывод ШИМа не подключен к выводу контроллера. Тикает внутри.

```
1 ISR(RTOS_ISR)
2 {
3 TimerService();
4 }
```

RTOS_ISR это тоже макроопределение. У меня оно привязано на **TIMER2_COMP_vect**, но можно в конфигах диспетчера привязать на что угодно. Сделано так опять же для того чтобы при переносе на другую архитектуру можно было пару строк подправить и все.

Постановщик таймеров
Функция устанавливающая задачу в очередь по таймеру. На входе адрес перехода (имя задачи) и время в тиках службы таймера — миллисекундах. Время двубайтное, т.е. от 1 до 65535. Если в очереди таймеров уже есть таймер с такой задачей, то происходит апдейт времени. Две одинаковых задачи в очереди таймеров не возможны. Это можно было бы реализовать, но на практике удобней апдейтить. Число таймеров выбирается исходя из одновременно устанавливаемых в очередь задач. Так как работа с глобальной очередью таймеров, то надо соблюдать атомарность добавления в очередь. Причем не тупо запрещать/разрешать прерывания, а восстанавливать состояние прерываний.

// Время выдержки в тиках системного таймера.

```
void SetTimerTask(TPTR TS, u16 NewTime)
{
u08  index=0;
u08  nointerrupted = 0;

if (STATUS_REG & (1<<Interrupt_Flag))     // Проверка запрета прерывания, аналогично функции выше
        {
        Disable_Interrupt
        nointerrupted = 1;
        }

for(index=0;index!=MainTimerQueueSize+1;++index)      //Прочесываем очередь таймеров
        {
        if(MainTimer[index].GoToTask == TS)           // Если уже есть запись с таким адресом
        {
        MainTimer[index].Time = NewTime;       // Перезаписываем ей выдержку
        if (nointerrupted)  Enable_Interrupt   // Разрешаем прерывания (если не были запрещены).
        return;                          // Выходим. Раньше был код успешной операции. Пока убрал
        }
 }

// Алгоритм, в данном случае не очень оптимален. В прошлом цикле можно было запомнить
// положение первого Idle и сейчас не искать его.
for(index=0;index!=MainTimerQueueSize+1;++index) // Если не находим похожий таймер, то ищем любой пустой
        {
        if (MainTimer[index].GoToTask == Idle)
                {
                MainTimer[index].GoToTask = TS;        // Заполняем поле перехода задачи
                MainTimer[index].Time = NewTime;       // И поле выдержки времени
                if (nointerrupted)  Enable_Interrupt   // Разрешаем прерывания
                return;                                        // Выход.
                }
        }
// тут можно сделать return с кодом ошибки - нет свободных таймеров
}
```

Постановка задачи по таймера идет следующим образом:

```
SetTimerTask(Task1,1000);
```

Вот и все.

Инициализация диспетчера:
В отличии от флагового автомата, где только флаги установить, тут требуется при старте сделать инициализацию очередей и очистку таймеров.
Функция выполняется один раз, потому инлайновая:

```
1  inline void InitRTOS(void)
2  {
3  u08   index;
4
5  for(index=0;index!=TaskQueueSize+1;index++)    // Во все позиции записываем Idle
6          {
7          TaskQueue[index] = Idle;
8          }
9
10 for(index=0;index!=MainTimerQueueSize+1;index++) // Обнуляем все таймеры.
11         {
12         MainTimer[index].GoToTask = Idle;
13         MainTimer[index].Time = 0;
14         }
15 }
16
17 //Потом роисходит запуск диспетчера. Собственно запускать то там надо таймер.
18 //RTOS Запуск системного таймера
19
20 //System Timer Config
21 #define Prescale               64
22 #define TimerDivider   (F_CPU/Prescaler/1000)      // 1 mS
23
24 inline void RunRTOS (void)
25 {
26 TCCR2 = 1<<WGM21|4<<CS20;      // Freq = CK/64 - Установить режим и предделитель
27                                 // Автосброс после достижения регистра сравнения
28 TCNT2 = 0;                     // Установить начальное значение счётчиков
29 OCR2  = LO(TimerDivider);      // Установить значение в регистр сравнения
30 TIMSK = 0<<TOIE0|1<<OCIE2;    // Разрешаем прерывание - запуск диспетчера
31
32 sei();
33 }
```

Главный файл, собственно файл проекта, выглядит так:

```
1  #include <HAL.h>
2  #include <EERTOS.h>
3
4  //RTOS Interrupt
5  ISR(RTOS_ISR)
6  {
7  TimerService();
8  }
9
10 // Прототипы задач ========================
11 void Task1 (void);
12 void Task2 (void);
13 void Task3 (void);
14 //=====================================
15
16 // Область задач
17 // Задачки простые, диодиком помигать. Одна зажигает, другая гасит.
18 // Зацикливаются они вызовом через диспетчер.
19 void Task1 (void)
20 {
21 SetTimerTask(Task2,100);       //Запускаем вторую задачу
```

```
22 LED_PORT  ^=1<<LED1;    // Зажигаем диодик
23 }
24
25 void Task2 (void)
26 {
27 SetTimerTask(Task1,100);        // Запускаем первую задачу
28 LED_PORT  &= ~(1<<LED1);        // Гасим диод.
29 }
30
31 // Результатом стал цикл из двух задач. Одна запускает другую по таймеру.
32 // первая зажигает диод, вторая гасит. В результате он мигает.
33 // А чтобы добавить, например, сканирование клавы мы добавляем еще одну задачу
34
35 void KeyScan()
36 {
37 SetTimerTask(KeyScan,50);       // И зацикливаем ее через диспетчер саму на себя
38 Scan();                         // Делаем полезную вещь.
39 }
40
41 int main(void)
42 {
43 InitAll();      // Инициализируем периферию
44 InitRTOS();     // Инициализируем ядро
45 RunRTOS();      // Старт ядра.
46
47 // Запуск фоновых задач. Для того чтобы задача завертелась кто то должен
48 // Запустить ее вручную хотя бы раз.  Делаем это перед главным циклом.
49
50 SetTask(Task1);
51 SetTask(KeyScan);
52
53 while(1)        // Главный цикл диспетчера
54 {
55 wdt_reset();    // Сброс собачьего таймера
56 TaskManager(); // Вызов диспетчера
57 }
58
59 return 0;
60 }
```

Прикладываю проект с уже поднятым диспетчером на базе ATMega16.

Там все раскидано по файликам.

```
1  GCC_RTOS.c              Это главный файл проекта.
2
3  EERTOS.c                Файл с ядром диспетчером.
4  EERTOS.h                Заголовочный файл ядра
5
6  EERTOSHAL.h             Файлы аппаратной абстракции и конфигурации ядра.
7  EERTOSHAL.c             Там я описал работу с прерывания для AVR  Файл с функциями
8                          аппаратной абстрацкии ядра. Запуск диспетчера.
9
10 HAL.c           Аппартная абстракция проекта. По возможности стараюсь сделать так, чтобы
11 все
12 HAL.h           аппаратно зависимые фишки были описаны в отдельном файле.
13                 В главном проекте только  основной алгоритм.
                   Не зависящий от типа контроллера.
```

Разгорелась целая дискуссия по поводу оптимальности как подхода в целом, так и конкретных реализаций. Есть много здравых мыслей. Я, в свою очередь, не претендую на оптимальность своего решения (порой бывает избыточной такая конструкция), но получилось достаточно удобно и когда надо быстро скреативить прошивку, да

так чтобы отлаживать не пришлось почти, то эта конвеерная молотилка как нельзя кстати приходится. Плюс тут очень многое поддается оптимизации. Можно сделать очередь на указателях в виде кольцевого буффера, можно оптимизировать таймер. Например, сделав его статичным.

В ассемлерных программах у меня тоже используется похожая структура. Правда переход идет не по реальным адресам, а по индексам в таблице переходов. Впрочем, в ранних постах я эту систему подробно описывал и все последующие примеры кода были уже на ней.

Файл с примером кода на диспетчере [1]

AVR. Учебный курс. Архитектура Программ. Часть 3

Приоритетный диспетчер.
Одной из проблем простого диспетчера является то, что все задачи имеют равный приоритет. С одной стороны, это просто и удобно. С другой — какое-либо важное событие можно прошляпить, пока там конвейер перещёлкает все задачи…

Проблему решает введение приоритетов.
В простейшем случае, можно ввести два приоритета — высокий и низкий. Разница между ними будет лишь в том с какой стороны очереди они будут засовываться на конвейер. Высокоприоритетные пихаются сразу в начало, низкоприоритетные с конца.
Разумеется, тут надо следить за тем, чтобы высокоприоритетные задачи не забивали конвейер, блокируя низкоприоритетные. Никакой защиты от этого нет, только думать головой.

Если нужна высокоуровневая система приоритетов, то можно очередь задач превратить в двумерный массив, где вторым этажом будет идти приоритет задачи. Правда при этом увеличиться время обработки конвейера — ведь надо будет сперва прочесать всю очередь в поисках наибольшего элемента. Но тут можно напридумывать кучу оптимизаций. Например, сортировать очередь при постановке задачи на конвейер, либо завести TOP list приоритетов, занося туда значения приоритетов. Тогда диспетчер, обрабатывая очередь, сразу будет искать нужный элемент, ориентируясь по TOP листу. Но вот так, на вскидку, я не берусь сказать какой из приемов будет эффективней/компактней/быстрей.

Кооперативная RTOS

На самом деле, называть RTOS такую систему нельзя. Она не является чисто реалтаймовой. Но название устоялось.

Это еще более сложная структура разделения кода. Отличается тем, что теперь у нас задания это не цепочки процедурок объединенных конвейером/флаг-автоматом, а полноценные законченные задачи.

Вот, например, программа мигания двумя светодиодами с разной частотой, на диспетчере (здесь и далее псевдокод), только функциональная часть:

```
1  Led1_OFF(void)
2  {
3  SetTimerTask(Led1_ON,1000);
4  led1_off();
5  }
6
7  Led1_On(void)
8  {
9  SetTimerTask(Led1_OFF,1000);
10 led1_on();
11 }
12
13 Led2_OFF(void)
14 {
15 SetTimerTask(Led2_ON,1000);
16 led2_off();
```

```
17 }
18
19 Led2_On(void)
20 {
21 SetTimerTask(Led2_OFF,1000);
22 led2_on();
23 }
```

Как видишь, у нас тут нет функционально законченных блоков. Все наши задачи представляют собой набор простейших действий залинкованных в цепочки друг на друга. Либо напрямую, либо через таймер. В очень больших программах, даже при такой абстракции запутаться уже несложно.

На кооперативной RTOS все будет выглядеть куда прозрачней:

```
1  Blink1_Task(void)
2  {
3  while(1)
4          {
5          Led1_on();
6          Dispatch(delay,1000);
7          led1_off();
8          Dispatch(delay,1000);
9          }
10 }
11
12 Blink2_Task(void)
13 {
14 while(1)
15         {
16         Led2_on();
17         Dispatch(delay,1000);
18         led2_off();
19         Dispatch(delay,1000);
20         }
21 }
```

Как видишь, программа двух действий разбилась на две независимые части. Мигание диодом 1 и мигание диодом 2.

Вырви из кода один блок, замени Dispatch(delay,1000) на delay_ms(1000); и задачу можно заливать без какой либо организации прям в контроллер и она будет работать как простейшая программа!
Т.е. теперь у нас уже получаются полноценные программы внутри основной программы. А что же такое

```
1 Dispatch(delay,1000);
```

А это команда API ядра RTOS. В нашем псеводокоде, допустим, ей мы приказываем передать управление диспетчеру (чтобы он занялся другими задачами) и вернуться сюда же когда таймерная служба натикает нам 1000мс.

Таким образом появляется многозадачность. Т.е. все неиспользуемое процессорное время (задержки) мы отдаем диспетчеру, который распределит его между другими задачами.

Таким образом, у нас получается что задача может быть активной — получит управление сразу же как это будет доступно, ожидающей чего-либо, например, того же таймера и неактивной — просто висящей в памяти программ. Сюда же могут быть добавлены приоритеты, а также чайный клуб с шахматами и поэтессами.

Задача может ожидать события от чего угодно — таймера,установки/сброса бита, сообщения от другой задачи. Также она может посылать события другим задачам.

Подробней расписывать не буду, о работе корпоративной RTOS очень хорошо расписано в **Salvo RTOS User Manual** (перевод Андрея Шлеенкова). [1]

А несколько позже, наверное, дам подробный раскур какой-нибудь кооперативной RTOS (пока еще даже не решил какой. Salvo, SCM или AVRX — сам еще ни одну из них толком не щупал).

Вытесняющая RTOS
Все не относящееся к вытесняющим ОС назвать полноценной операционной системой **реального времени** нельзя. Дело в том, что понятие **реального времени** означает, что чтобы ни случилось — время обработки и создание реакции на критического событие не превысит заданного порога.

Т.е. если заявлено что при ядерной ялярме стержни в реактор упадут в течении 50мс, то должно быть так.

Сам понимаешь, что ни диспетчер, ни кооперативная ОС этого гарантировать не могут. Если ты где-то не передашь управление диспетчеру (ну ошибся, бывает) и у тебя проц зациклиться в одной задаче, то повиснет вся система. Реальное время тут гарантируется только мозгами разработчика, четко понимающего что он творит.

Этого недостатка лишена вытесняющая RTOS. Она является по настоящему реалтаймовой.

И задачи там могут быть вида:

```
1  Blink1_Task(void)
2  {
3  while(1)
4          {
5          Led1_on();
6          delay_ms(1000);
7          led1_off();
8          delay_ms(1000);
9          }
10 }
11
12 Blink2_Task(void)
13 {
14 while(1)
15         {
16         Led2_on();
17         delay_ms(1000);
18         led2_off();
19         delay_ms(1000);
20         }
21 }
22
23 Да хоть
24
25 DumbLoop(void)
26 {
27 m1: goto m1
28 }
```

Тут это не важно. Диспетчер тут обладает полномочиями прерывать выполнение любой задачи когда угодно, между двумя любыми командами процессора — делается это посредством прерывания на котором сидит диспетчер. В результате, каждая задача выполняется рывками, кусочками заданного интервала. Скажем, по 10мс на задачу. В этом случае, даже если одна будет наглухо висеть, то вся система не повиснет, т.к. диспетчер будет у ней забирать управление и делать более важные вещи. Круто, да?

Но за крутизну приходится платить. Т.к. у нас происходит отбирание управления у одной задачи и передача другой, то надо сохранять состояние остановленной задачи. И одним сохранением регистров (как это было в прерывании) тут не обойдешься. Ведь у каждой задачи будет еще и собственный стек, и локальные переменные и регистры.
В результате, на каждую задачу выделяется свое адресное пространство с запасом и не дай боже одной задаче потереть контекст другой — все рухнет как карточный домик.

На восьмиразрядных микроконтроллерах вытесняющие RTOS существуют (правда название не вспомню, но видел), однако худо бедно этот монстр может запуститься разве что на ATmega128. Да и то большая часть

ресурсов, в первую очередь оперативная память, МК будет занята проворачиванием диспетчера, так что кооперативки тут предпочтительней. А вот на ARM ситуация с этим делом уже намного лучше.

AVR. Учебный Курс. Архитектура Программ. Часть 4

Вытесняющий диспетчер
Давным-давно, когда я учился в школе, мне не давал покоя вопрос. Как работают параллельные операционки? Как тот же самый Windows умудряется переключать процессы, не терять регистры (да, я тогда уже начинал учить асму), как он определяет момент переключения, почему все это работает? Виртуальная память, проверка на ошибочный код — никто ничего этого не объяснял. Все твердили про какие-то там объекты, классы, и говорили очень виртуально. А мне мясо, мясо давай!

Но вот прошло время, и я начал писать программы под МК. Тут я получил полную свободу действий и абсолютную прозрачность кода. И почти сразу я стал писать свою операционку. Не для того, чтобы что-то запускать на ней, а лишь для того, чтобы по ходу понять и сообразить, как же все это устроено. И эта статья как раз о том, что я узнал.

Отмазки
Сея программа не более чем учебный пример. Т.к. не тестировалась в серьезных условиях. Не использовалась в каких либо проектах, а гонялась только на эмуляторе. Смысл ее не дать готовое решение, а показать принцип и механизм работы диспетчера операционной системы с принудительным переключением задач. Да и AVR это не та платформа на которой имеет смысл городить вытесняющие диспетчеры.

Немного теории
Думаю не секрет, что любая многозадачная операционка работает на прерываниях таймера. Даже на x86 архитектуре без него никуда.

При срабатывании прерывания процессор идет на обработчик, а там прописано примерно так:

- 1) Сохранение регистров.
  - 1.1) Сохранение виртуальной памяти — в файл подкачки или еще куда-нибудь.
- 2) Сохранение текущего адреса.
- 3) Загрузка регистров след. процесса.
  - 3.1) Загрузка виртуальной памяти.
- 4) Загрузка адреса возврата.
- 5) Переход по этому адресу.

По сути — не так уж и много работы, если исключить пункты X.Y. Всего и делов — сохранить, загрузить и вернуться. Вот именно это я и реализовал — простейшее ядро операционки, БЕЗ проверок и т.д.

Структура моей операционки

Каждый процесс имеет собственную память — 64 байта (число взял от балды, просто чтобы красивое было): 35 байт — место для сохранения регистров и SP, остальные 29 — под стек.

Да, кстати: переключение процессов дополнительно нагружает стек на 3 байта. Поэтому больше 26ти байт лучше не использовать, или же выделять CLI/SEI.

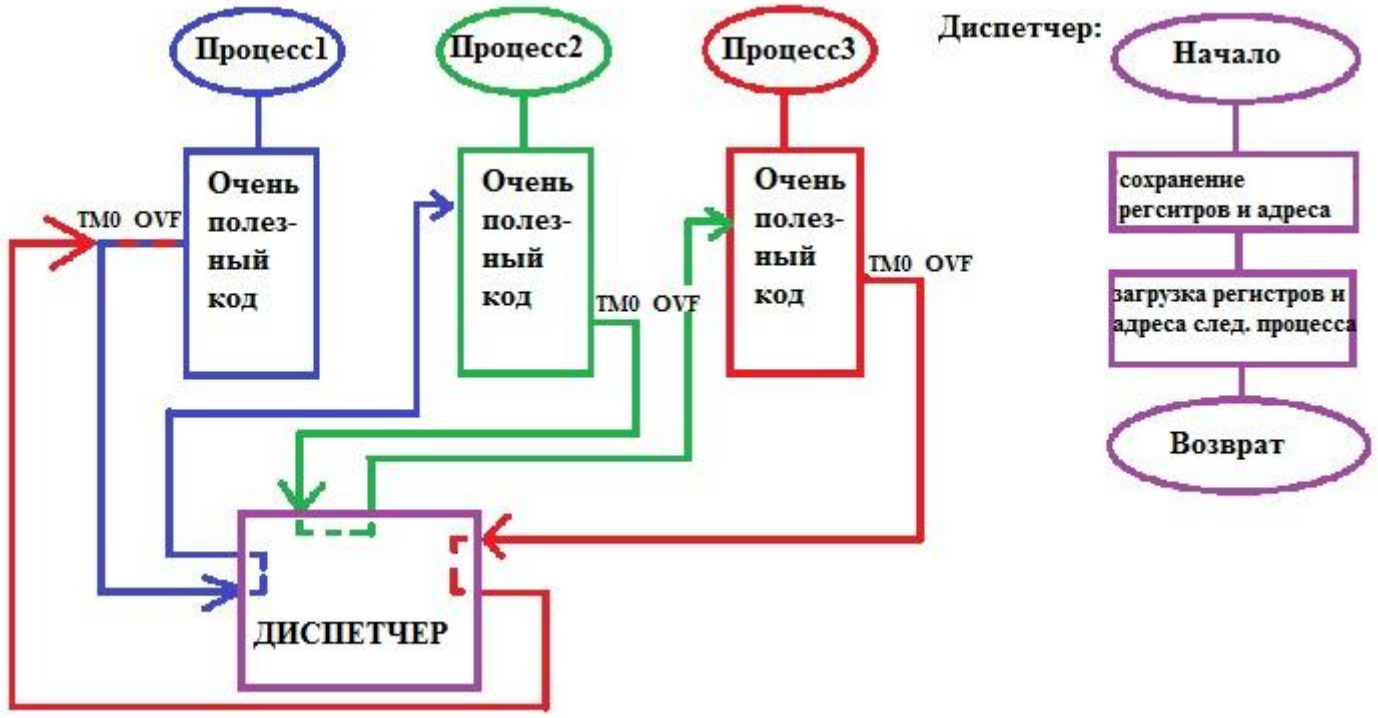
В памяти находится очередь процессов — она начинается с нуля (нулевой, ущербный процесс), и заканчивается FF. Почему FF? А просто так удобнее отлаживать: проще найти очередь в памяти в эмуляторе. Это единственная причина, поэтому можно его убрать. С нулевым процессом нельзя проводить никакие махинации — никаких пауз и таймеров к нему не использовать, иначе это приведет к апокалипсису… Наверное. Не пробовал.

Каждому процессу соответствует свой таймер. Это двухбайтные числа, лежащие в массиве, устанавливаются при помощи «API» функции SetTimer.Используется, чтобы временно убрать процесс из очереди.

Работа диспетчера
Когда наступает прерывание переполнения таймера — включается диспетчер. Работает он на процессоре в 8МГЦ 58.5 микросекунд, так что не мешает ни передачам данных (если конечно скорость не запредельная), ни работе

процессов. Диспетчер сохраняет все регистры текущего процесса, адрес возврата, пересчитывает таймеры, запускает новые процессы (если их таймер дотикал до нуля), после чего загружает регистры следующего процесса из очереди, достает его адрес возврата и прыгает туда.

Схематично можно изобразить так:



Код

Примечание: крупные куски кода я распихал в макросы по смыслу — иначе функция становилась абсолютно нечитаемой, разобраться было невозможно. Поэтому не надо хвататься за голову и искать макросы — названия все говорят сами за себя, а устройство потом опишу.
В коде полно мусора. Я это знаю. Писал операционку давно, много раз апгрейдил, переделывал, оттуда столько и осталось.

Начну с подробного описания переменных.
В суть этой части переменных можно не вдаваться — это всего лишь временные переменные. Весь прикол в том, что при переключении SP тоже меняется, и использование стека становится до ужаса неудобным. К тому же прогружать и без того некрупный стек не стоит.

```
.equ ProcDataSize = 64          ; память процесса
.equ ProcNum = 8                ; колчиество процессов

; Куча временных переменных, по большей части для перекидывания данных
TempProc:       .byte 1         ; для распознания
SREGT:          .byte 1         ; временная переменная под SREG
SPTL:           .byte 1         ; под SPL
SPTH:           .byte 1         ; под SPH
R16T:           .byte 1         ; под R16 :)
ZLT:            .byte 1         ; ZL
ZHT:            .byte 1         ; ZH
ADDRL:          .byte 1         ; адрес
ADDRH:          .byte 1
TCurp:          .byte 1         ; временный CurProc (см. ниже)
```

Дальше переменные очереди:

- TasqQueue — очередь процессов. В ней находятся номера процессов в определенном порядке.
- LastItem — «указатель» на последний элемент очереди (именно поэтому FF в конце очереди не используется: через указатель намного удобнее работать).
- CurProc — текущий номер процесса. Используется при переключении и в SuspendProcess.
- TimerQueue — МАССИВ таймеров. Да-да, именно массив. Саначала я хотел делать очередь, но потом решил, что массив проще в реализации. А имя осталось старым.
- Proctable — таблица процессов. В нее при старте заносятся адреса всех процессов, чтобы далее работать не с адресами, а с номерами, что значительно упрощает программу.

```
                                        ; Очередь и все, что с ней связано
CurProc:        .byte 1                         ; Системная переменная, хранит номер процесса
LastItem:       .Byte 1                         ; Номер последнего элемента очереди (для
удобства)
TaskQueue:      .byte ProcNum+1                 ; Очередь процессов
TimerQueue: .byte ProcNum*2 ; Массив таймеров

Proctable:      .dw 1                   ; Таблица адресов процессов
                .dw 1                   ; для наглядности - 8 процессов отдельно
                .dw 1
                .dw 1
                .dw 1
                .dw 1
                .dw 1
                .dw 1
```

И, пожалуй, самая важная часть операционки — место под данные процессов. То самое, что описано в теории.

```
; Самое важное: место под стек и регистры.
; Сколько процессов - столько и этих записей
ProcData:       .byte ProcDataSize
                .byte ProcDataSize
                .byte ProcDataSize
                .byte ProcDataSize
                .byte ProcDataSize
                .byte ProcDataSize
                .byte ProcDataSize
                .byte ProcDataSize
```

Диспетчер.

Код диспетчера, как ни странно, прост. К тому же я его прокомментировал достаточно подробно. Пробегись глазами.

```
TM0_OVF:
        STS R16T, R16                   ; Выковыриваем из стека адрес
        POP R16                         ; в переменные ADDRx
        STS ADDRL, R16                  ; предварительно сохранив регистр R16
        POP R16
        STS ADDRH, R16
        LDS R16, R16T

        SaveRegs CurProc                ; Сейвим регистры

        SaveAddr CurProc                ; Сохраняем адрес возврата

        DecTimers                       ; Декаем таймеры

        GetNextProc                     ; Берем следующуий процесс

        STS ADDRL, ZL                   ; Сейвим адрес в ADDRx
        STS ADDRH, ZH
```

```
        loadRegs                        ; Грузим регистры

        STS ZLT, ZL                     ; сохраняем Z, лишний раз не грузим стек
        STS ZHT, ZH
        LDS ZL, ADDRL                   ; ADDRx в Z
        LDS ZH, ADDRH
        PUSH ZL                         ; Пишем в стек адрес возврата
        PUSH ZH
        LDS ZL, ZLT                     ; и респим регистры Z
        LDS ZH, ZHT

        STS R16T,R16                    ; сейвим R16
        LDS R16, SREGT                  ; восстанавливаем SREG. ДАЛЕЕ НИКАКОЙ МАТЕМАТИКИ!!!
        OUT SREG, R16                   ; а то испортим
        OUTI TCNT0, 0                   ; Обнуляем таймер
        OUTI TIFR, 1                    ; и его флаг прерывания
        LDS R16, R16T                   ; вернем многострадальный R16
        RETI                            ; Валим нафиг
```

Первое, что я делаю — достаю из стека адрес возврата и сохраняю в переменные. Зачем? А затем, что после некоторых манипуляций стек переключится на другой процесс, а там будут уже совершенно другие данные. Сохранится он потом в ProcTable. Сейчас ВРОДЕ БЫ код написан так, что это не обязательно. Однако так надежнее, и, если что, проще дописать новые возможности.

Далее пресловутое сохранение регистров, адреса, пересчет таймеров, переход к следующей процедуре в очереди (доставание ее адреса), загрузка регистров нового процесса. Под конец пишем в стек НОВЫЙ адрес возврата, для другого процесса, и RETI прыгает туда.

Обрати внимание, что восстановление SREG из переменной идет в самом конце диспетчера. Иначе при первой же математической операции (будь то Add, Sub или даже INC) испортятся флаги. А я этого не хочу. Также в конце сбрасывается таймер — одному Богу известно, сколько он там натикал за время работы диспетчера, и сколько осталось до следующего вызова.

Чтож, настал час, когда я покажу тебе самое сложное для восприятия во всей операционке — макросы. Вкратце со смыслом каждого думаю ты ознакомился, прочитав комментарии к коду выше (или нет?).
3…2…1… Поехали!

```
; Макрос сохранения регистров процесса
        .macro SaveRegs
                STS  ZLT, ZL            ; Спасаем регистры
                STS  ZHT, ZH            ; стараемся как можно меньше использовать стек
                STS  R16T, R16
                IN       R16, SREG
                STS  SREGT, R16

                LDS  R16, TempProc              ; Определяем, где этот макрос применен
                CPI      R16, $FF               ; В SuspendProcess или в переключении
                BRNE NoLoadNum

                        LDI ZL, low(TaskQueue) ; Получаем номер процесса из очереди
                        LDI ZH, High(TaskQueue)
                        LDS R16, @0
                        ADD ZL, R16
                        LDI R16, 0
                        ADC ZH, R16
                        LD  R16, Z

                NoLoadNum:
                STS TCurP, R16                  ; И сохраняем в переменную

                LDI ZL, low(Procdata)           ; Получим адрес памяти процессора:
                LDI ZH, High(Procdata)
```

```
        Push R17                        ; Снова спасаем регитсры
        Push R0
        Push R1
        LDS R16, TCurP                  ; Получаем адрес памяти процесса
        LDI R17, ProcDataSize
        MUL R16, R17
        ADD ZL, R0
        ADC ZH, R1
        Pop R1                          ; Вернем регистры
        Pop R0
        Pop R17
        LDS R16, R16T

        ST      Z+,R0                   ; Долго и нудно их сейвим
        ST      Z+,R1
        ST      Z+,R2
        ST      Z+,R3
        ST      Z+,R4
        ST      Z+,R5
        ST      Z+,R6
        ST      Z+,R7
        ST      Z+,R8
        ST      Z+,R9
        ST      Z+,R10
        ST      Z+,R11
        ST      Z+,R12
        ST      Z+,R13
        ST      Z+,R14
        ST      Z+,R15
        ST      Z+,R16
        ST      Z+,R17
        ST      Z+,R18
        ST      Z+,R19
        ST      Z+,R20
        ST      Z+,R21
        ST      Z+,R22
        ST      Z+,R23
        ST      Z+,R24
        ST      Z+,R25
        ST      Z+,R26
        ST      Z+,R27
        ST      Z+,R28
        ST      Z+,R29

        MOV YL, ZL                              ; Перекидываем адрес в Y пару
        MOV YH, ZH
        LDS ZH, ZHT                             ; Достаем Z
        LDS ZL, ZLT
        ST  Y+, R30                             ; И сейвим его
        ST      Y+, R31

        LDS R16, SREGT                          ; Сейвим SREG
        ST  Y+, R16
        IN      R16, SPL                        ; и Stack Pointer
        ST      Y+, R16
        IN      R16, SPH
        ST      Y+, R16
.endm
```

В самом начале я использую временные переменные для сохранения регистров, чтобы лишний раз не нагружать стек. Потом идет распознание, откуда его вызывали по переменной TempProc.

— Почему бы не использовать параметры макроса? — спросишь ты.
— А потому что переключение процессов иногда вызывается по прерыванию, а иногда — в функции SuspendProcess. Получается, что макрос вызывается из одного и того же места, но «это место» срабатывает по разным причинам:) и надо как-то это отслеживать. — отвечу я.
Если TempProc = $FF — значит вызывали переключение по прерыванию, тогда выполняется полный код: т.е. получаем номер процесса по текущему индексу элемента в очереди. Если же нет — то не надо загружать номер процесса. Это для SuspendProcess.

Сейвим регистры в стек — как раз те самые 3 байта. Здесь можно использовать и временные переменные, но все равно макрос загрузки регистров грузит стек на 3 байта. Можно и там использовать переменные, но код становится совсем нечитаем, даже относительно того, чтоо счас. К тому же это тоже остатки мусора.
Как получаю адрес памяти процесса — беру номер, умножаю на размер 1й ячейки (64 байта) и получаю смещение. Прибавляю адрес первой и получаю то, что надо.

И вот мы дошли до самого сохранения регистров. Я думаю, понятно, как оно происходит до R29 :) Чтобы сохранить Z пару, делаю просто: Y пару менять не страшно, ведь Y уже сохранен. Поэтому перекидываю туда адрес, лежащий в Z, и сохраняю Z. Достаю заготовленный в самом начале макроса SREG и пишу в память. И наконец сохраняю стек.

Процедура загрузки похожа на сохранение, только все с точностью до наоборот :)

```
; Макрос загрузки регистров процесса

        .macro LoadRegs
               LDI ZL, low(TaskQueue)                ; Грузим номер процесса из очереди:
               LDI ZH, High(TaskQueue)                       ; В Z - адрес очереди
процессов
               LDS R16, CurProc                      ; В R16 - номер текущего процесса в
очереди
               ADD ZL, R16                           ; Прибавляем его к адресу
               CLR R16
               ADC ZH, R16                           ; и грузим реальный номер
               LD  R16, Z

               LDI ZL, low(Procdata)                 ; Вычисляем адрес данных процесса
               LDI ZH, High(ProcData)

               LDI R17, ProcDataSize                 ; Address = ProcData + ProcDataSize*N
               MUL R16, R17                          ; где N = R16, только что грузили
               ADD ZL, R0
               ADC ZH, R1

               LD      R0, Z+                        ; Респим регистры
               LD      R1, Z+
               LD      R2, Z+
               LD      R3, Z+
               LD      R4, Z+
               LD      R5, Z+
               LD      R6, Z+
               LD      R7, Z+
               LD      R8, Z+
               LD      R9, Z+
               LD      R10, Z+
               LD      R11, Z+
               LD      R12, Z+
               LD      R13, Z+
               LD      R14, Z+
               LD      R15, Z+
               LD      R16, Z+
               LD      R17, Z+
               LD      R18, Z+
               LD      R19, Z+
               LD      R20, Z+
```

```
                LD      R21, Z+
                LD      R22, Z+
                LD      R23, Z+
                LD      R24, Z+
                LD      R25, Z+
                LD      R26, Z+
                LD      R27, Z+
                LD      R28, Z+
                LD      R29, Z+

                PUSH YL                                 ; Сейвим Y пару
                PUSH YH
                MOV YL, ZL                              ; Перекидываем туды Z
                MOV YH, ZH
                LD      R30, Y+                         ; И восстанавливаем Z
                LD      R31, Y+

                PUSH R16                                ; Сейвим R16
                LD  R16, Y+
                STS SREGT, R16                          ; Грузим SREG
                LD      R16, Y+                         ; ВНИМАНИЕ, ИЗВРАТ!
                                                        ; (Ну почему изврат. Нормальные будни
ассемблерщика прим. DI HALT)
                STS     SPTL, R16                       ; Сохраняем SP в переменные
                LD      R16, Y+                         ; это будет использоваться далее
                STS  SPTH, R16                          ; делается потому, что нельзя здесь
                                                        ; менять SP - в стеке наши регистры

                POP R16                                 ; Откапываем в стеке сокровища

                POP YH
                POP YL

                STS R16T, R16                           ; Сейвим R16 в переменную
                LDS R16, SPTL
                OUT SPL, R16                            ; Из переменных для SP грузим SP
                LDS R16, SPTH
                OUT SPH, R16
                LDS R16, R16T                           ; И возвращаем R16
                                                        ; Всё, изврат закончился.
        .endm                                           ; Итог: восстановили все регистры и SP
```

Здесь уже с предварительным сохранением регистров можно не париться. Мы же их все равно загружаем :) Начало то же самое, что и в сохранении после проверки TempProc. Загружаем все регистры, так же перекидываем Z в Y и т.д. А вот в конце интересная вещь: если загружать значения SP прямо по ходу, то получится, что Y положится в стек одного процесса, а вынется из стека другого! Естественно, это будет полный бред. Поэтому гружу значения SP в переменные, достаю Y и R16, и только потом переношу в SP.

Поздравляю, мы уже на пол пути :)

```
; Макрос сохранения адреса возврата в таблицу
        .macro SaveAddr
                LDS  R16, TempProc                      ; Определим, откуда вызвали
                CPI      R16, $FF
                BRNE NoLoadNumA

                        LDS R16, @0

                        LDI ZL, low(taskQueue)          ; берем номер процесса из очереди
                        LDI ZH, High(taskQueue)

                        ADD ZL, R16
                        LDI R16, 0
```

```
                ADC ZH, R16

                LD R16, Z
        NoLoadNumA:

        LDS YL, ADDRL                           ; грузим засейвенный адрес
        LDS YH, ADDRH

        LDI ZL, low(Proctable)                  ; И сейвим его!
        LDI ZH, High(ProcTable)

        LSL R16
        ADD ZL, R16
        LDI R16, 0
        ADC ZH, R16

        ST Z+, YH
        ST Z+, YL
.endm
```

Все просто, только много кода. Если вызывается из диспетчера — то достаем номер процесса из очереди, идем в таблицу и пишем туда наш адрес из ADDRx. О SuspendProcess — позже.

Идем дальше:

```
; Макрос получения след. процесса по очереди-------------------

        .macro GetNextProc
                LDI R16, $FF
                STS TempProc, R16               ; В TempProc пишем $FF

                LDS R16, CurProc                ; Инькаем CurProc
                INC R16

                LDS R17, LastItem               ; Сравниваем с последним
                INC R17
                CP  R16, R17
                BRCS NoDec                      ; Если больше - пишем 0

                        LDI R16, 0

        NoDec:
                STS CurProc, R16                ; сохраняем в CurProc

                LDI ZL, low(taskQueue)          ; Грузим номер из очереди...
                LDI ZH, High(taskQueue)                 ; (было выше)

                ADD ZL, R16
                LDI R16, 0
                ADC ZH, R16

                LD  R16, Z

                LDI ZL, low(ProcTable)          ; Берем из таблицы адрес процесса...
                LDI ZH, High(ProcTable)

                LSL R16
                ADD ZL, R16
                LDI R16, 0
                ADC ZH, R16

                LD R16, Z+
                LD R17, Z+
```

```
        MOV ZL, R16                     ; И пихаем его в Z!
        MOV ZH, R17
.endm
```

Тут тоже все просто до жути. Увеличиваем CurProc, если ушло за конец очереди — пишем 0, вынимаем из очереди номер процесса, идем в таблицу, достаем адрес возврата и пишем его в Z пару.

И последний макрос:

```
; Макрос пересчета таймеров
.macro DecTimers
        LDI ZL, Low(TimerQueue)         ; Грузим начало очереди таймеров
        LDI ZH, High(TimerQueue)

        LDI R16, 0
Decrease:                               ; Цикл уменьшения таймеров
        LD  R17, Z+                     ; Таймеры 2х-байтные
        LD      R18, Z
        CPI R17, 0                      ; побайтово сравниваем
        BRNE Decr                       ; Если 0 - то не надо уменьшать
        CPI R18, 0
        BREQ NoDecrease
Decr:
         SUBI R17, 1                    ; уменьшаем на 1
         SBCI R18, 0
         LD R0, -Z                      ; и переписываем старые значения
         ST Z+, R17
         ST Z,  R18
         CPI R17, 0                     ; если стало равно нулю - надо
         BRNE NoDecrease                ; запускать процесс
         CPI R18, 0
         BRNE NoDecrease

          MOV R17, R16                  ; в R17 номер процесса
          RCALL StartProcess            ; API StartProcess

NoDecrease:
        LD R0, Z+                       ; переход к следующей ячейке
        INC R16                         ; Inc R16; если стал равен кол-ву
        CPI R16, ProcNum                ; процессов - значит выход.
        BRNE Decrease
.endm
```

Тут мы проходимся по всему массиву таймеров, уменьшаем все на 1 (если они не равны нулю конечно), и, если таймер достиг нуля, то запускаем соответствующий ему процесс. Алгоритм думаю достаточно прокомментирован.

Да, вспомнил. Таймер в моей операционке — далеко не точный. Как понятно из кода, число в таймере — количество переключений процессов до следующего запуска. Его можно использовать лишь чтобы например примерно каждые 20 миллисекунд параллельно основной программе сканировать клавиатуру, каждые 40мс обновлять дисплей и т.п. Вообщем там, где точность не важна. По моим подсчетам точность варьируется в пределах 0.1мс.

Вот и вся операционка. Если вдуматься, ничего сложного тут нет. В скомпилированном виде вместе с АПИ (см. ниже) весит 1190 байт. Для вытесняющей операционки немного :)

Если ты до сих пор не устал, то сейчас я покажу тебе API.

Application Programming Interface (API) я сделал из 5ти функций. Просто чтобы было легко потом дописать загрузку процесса например из EEPROM или еще чего. Можно конечно использовать его в своих целях.

CreateProcess:
функция вызывается так:

```
        LDI R17, N
        RCALL CreateProcess
```

N — номер процесса в таблице Proctable (помни, адреса давно остались в прошлом, у нас номера:))
Сама функция такая:

```
;Создание нового процесса
 CreateProcess:                   ; R17 - номер процесса в таблице адресов
        LDS R16, LastItem     ; Увеличим номер последнего элемента
        INC R16
        CPI R16, ProcNum      ; Если больше размера очереди - выходим
        BREQ Exit
        STS LastItem, R16     ; увеличим конец очереди

; Инициализация нового процесса
        LDI YL, Low(ProcData) ; Начало области данных
        LDI YH, High(ProcData)

        INC R17
        LDI R16, ProcDataSize ; Получаем начало данных след. процесса
        MUL R16, R17
        ADD YL, R0
        ADC YH, R1
        DEC R17
        LD  R16, -Y           ; Уменьшим на 1 - конец данных нужного
                              ; В Y пару сохраняем этот адрес -
                              ; Это будет стек

        LDI ZL, Low(ProcData) ; Начало области данных
        LDI ZH, High(ProcData)

        LDI R16, ProcDataSize ; Получаем начало данных нужного процесса
        MUL R16, R17
        ADD ZL, R0
        ADC ZH, R1

        LDI R16, 33           ; Очищаем место регистров - 32 регистра,
        LDI R18, 0            ; флаг состояния =&gt; пишем 33 нуля
 Clear:
        ST  Z+, R18
        DEC R16
        BRNE Clear

        ST  Z+, YL            ; Пишем адрес стека
        ST      Z+, YH

; Запуск процесса  - ставим в очередь

        LDS R16, LastItem
        LDI ZL, Low(TaskQueue)               ; Загружаем  адрес в очереди
        LDI ZH, High(TaskQueue)       ; Исходя из номера последнего элемента

        ADD ZL, R16
        LDI R16, 0
        ADC ZH, R16

        ST Z+, R17            ; И пишем туда наш номер процесса,
        LDI R16, $FF          ; потом FF
        ST Z+, R16
Exit:
```

```
        RET
```

Сначала увеличим указатель на последний элемент очереди. Дальше очищаем память процесса. Адрес начала стека — это адрес начала данных следующего процесса минус единица. Чистим 33 байта для регистров (чтоб при старте у процесса все регистры обнулились, включая SREG), и вписываем после 33х нулей адрес стека процесса. Вот и проинициализировали (слово-то какое!). Пишем в конец очереди наш номер процесса — и все готово. Как хорошо работать с номерами: так бы пришлось постоянно возиться с двубайтными адресами, переписывать их в очереди и т.д. Вообщем, была бы полная неразбериха.

Следующая АПИ-функция:

```
; Запуск процесса после паузы
; Используется системой, но, думаю, можно и самому попробовать :)
 StartProcess:                          ; R17 - номер процесса
         PUSH ZL
         PUSH ZH
     PUSH R16
         LDS R16, LastItem              ; Увеличим номер последнего элемента
         INC R16
         CPI R16, ProcNum               ; Если больше размера очереди - выходим
         BREQ ExitS

         Push R17

         STS LastItem, R16
         LDI ZL, Low(TaskQueue)                 ; Далее надо сдвинуть очередь
         LDI ZH, High(TaskQueue)        ; что мы и делаем

         ADD ZL, R16                    ; получаем адрес конца очереди
         CLR R17
         ADC ZH, R17

         MOV R17, R16                   ; сдвигать надо от конца и до текущего
         LDS R18, CurProc               ; т.к. вставляем процесс после текущего
         SUB R17, R18                   ; В R17 - количесво сдвигов
 ShellS:                                        ; Сдвигаем (да, криво, я знаю)
                 LD R18, Z+
                 ST Z, R18
                 LD R0, -Z
                 LD R0, -Z
                 DEC R17
                 BRNE ShellS

         LD R0, Z+                      ; и пишем в освободившееся место интересующий
         POP R17                        ; номер процесса
         ST  Z, R17
ExitS:
        POP R16
        POP ZH
        POP ZL
        RET
```

Функция ставит уже созданный процесс в очередь после текущего. Как?

Увеличиваем указатель на последний элемент, как и в CreateProcess. Потом сдвигаем очередь — с текущего элемента и до конца, и пишем на освободившееся места нужный номер процеса. Надеюсь, это понятно.

Самая сложная функция: пауза процесса. Что для этого нужно?

Нужно убрать интересующий процесс из очереди. Вроде все просто. НО если мы убираем текущий процесс — надо немедленно перейти к следующему. А если другой — то ни в коем случае нельзя сохранять регистры. Это я и делаю:

```
;Поставить процесс на паузу
;Удаление - вечная пауза :)

; Внимание! Перед функцией ОБЯЗАТЕЛЬНО необходимо запретить прерывания,
; а после - разрешить. Иначе возможно будет плохо.

 SuspendProcess:                        ;R17 - номер процесса
        LDI ZL, Low(TaskQueue)                  ; начало очереди
        LDI ZH, High(TaskQueue)

        LDI R16, 0
 Seek:                                  ; Ищем процесс с таким номером в очереди
              LD R18, Z+
              CP R18, R17
              BREQ EOSeek               ; Нашли - идем дальше
              INC R16
              CPI R16, LastItem
              BRNE Seek

              RJMP NotFound             ; А если нет такого - выходим
 EOSeek:

        LDI R19, 0                      ; для распознания в макросе
        STS TempProc, R17
        MOV R17, R16                    ; Сохраняем номер в очереди
        LDS R16, CurProc
        CP      R16, R17                ; Если не равен текущему -
        BRNE TNoSave                    ; то не сохраняем регистры
 TSave:
               LDI R19, 1               ; тоже для распознания в дальнейшем
               RJMP NoDec               ; первое, что пришло в голову :)

 TNoSave:
               CP R16, R17              ; Если меньше текущего -
               BRCS NoDec               ; То уменьшаем номер текущего в очереди
                      LDS R16, CurProc
                      DEC R16
                      STS CurProc, R16

 NoDec:
        LDI ZL, Low(TaskQueue)                  ; Далее надо сдвинуть очередь
        LDI ZH, High(TaskQueue)         ; что мы и делаем

    INC R17                     ; От текущего и до конца
        ADD ZL, R17
        LDI R16, 0
        ADC ZH, R16
        DEC R17

        LDS R16, LastItem
        SUB R16, R17
        INC R16
 Shell:                                 ; Сдвигаем (да, криво, я знаю)
               LD R18, Z
               ST -Z, R18
               LD R0, Z+
               LD R0, Z+
               DEC R16
               BRNE Shell

        LDS R16, LastItem       ; уменьшаем номер последнего элемента
        DEC R16
        STS LastItem, R16
```

```
        CPI R19, 0
        BREQ NotFound          ; и если стопим текущий процесс - то переключим
          RJMP TM0_OVF

 NotFound:                     ; Иначе - с вещами на выход
       RET
```

Ищу в очереди процесс с нужным номером, определяю, сейвить регистры или нет. Потом сдвигаю очередь, уменьшаю указатель и в конце смотрю, что определил заранее: если текущий процесс останавливаем — то сохраняем регистры и переключаемся. Если нет — то переключать не надо, и сохранять регистры тем более.

Теперь рассмотрим таймеры:

```
;Поставить таймер

/*
 Внимание! данная функция должна использоваться вместе с Suspend porcess
 Иначе через установленную задержку запустится второй экземпляр процесса,
 имеющий тот же адрес, ту же память и тот же стек, НО работающий парал -
 лельно первому, Соответственно их регистры будут мешаться друг с другом,
 указатель стека будет прыгать туда-сюда - вообщем, не очень приятно.

 делается так:
 ...
 CLI
  Rcall SetTimer
  Rcall Suspend process
 STI
 ...
*/
 SetTimer:                              ;R17 - номер процесса, R18(L), R19(H)-задержка
        PUSH R16
        PUSH R17
        LDI ZL, Low(TimerQueue)        ; начало массива таймеров
        LDI ZH, High(TimerQueue)

        LDI R16, 0
        ROL R17                        ; умножим адрес на два - таймеры двухбайтные
        ROL R16                        ; а вдруг кто-то 129 процессов замутит:)
        ADD ZL, R17                    ; получаем адрес
        ADC ZH, R16

        ST      Z+, R18                ; Ставим таймер
        ST  Z+, R19                    ; Ставим таймер
        POP R17
        POP R16
       RET                             ; выход
```

Функция проста — ищем нужный элемент в массиве и пишем туда значения таймера. ВСЕ! Остальное будет делать диспетчер.
А теперь подумай: раз таймер измеряется в количестве переключений, разве может он точно работать? Может. НО только в случае, если ты НИГДЕ больше не используешь CLI/STI, SuspendProcess и таймеры других процессов. Потому что в первом случае (cli/sti) просто диспетчер может запоздать, во втором — тоже диспетчер опоздает, но уже намного (т.к. большой кусок кода в CLI/SEI), а в третьем — если несколько таймеров дотикают до нуля одновременно, то процессы поставятся в очередь по возрастанию, и неизвестно, когда очередь дойдет до твоего процесса.

Я считал — вроде можно поставить двумя байтами задержку до 5.5 секунд. Думаю, этого достаточно. Если хочешь — можно переписать под 3х байтовые таймеры. Тогда будет до 1408 секунд.

И самая простая АПИ из всех, состоит из одной команды :D

```
; Процедура "Перейти к след. процессу" ------------------------------
```

```
/*
Usage:
  надо, чтобы следующим процессом в очереди стал такой-то?
  Ну просто позарез надо? Да еще чтоб запустился немедленно?
  Тогда эта функция для вас!
  Делаем так:

  ...
  LDI R18, 1           ; Если задержка = 1 - процесс запустится
  LDI R19, 0           ; при следущем переключении. Нам это и надо.
  CLI
  RCALL SetTimer       ; Ставим таймер
  RCALL SuspendProcess ; Отключим тот процесс, который надо поставить - он уберется из
очереди
  RCALL GoToNextProc   ; А функция - типа API ^_^ при переключении таймер дотикает дло нуля
                       ; и поставит процеес следующим в очереди
  SEI                  ; чисто для приличия - RETI сам это сделает
*/
 GoToNextProc:
       CLI             ; Защита (на всякий)
       RJMP TM0_OVF    ; Валим на прерывание таймера (не пашет? убери R)
 RET                   ; Сюда вернется процесс, поэтому RET нужен.
```

Гениальнейшая функция :)

Вот и все. Я детально описал всю свою операционку. Оптимизацией почти не занимался, это думаю видно. Конечно, на каком-нибудь ATMEGA8/16 использовать такое не особенно нужно. Однако, как я писал выше, основной целью написания этой операционки было ознакомление с механизмом переключения процессов. Искренне надеюсь, что ты понял все, что я писал.

На реальном девайсе я это ни разу не запускал, работал только с тремя процессами в эмуляторе. Там ничего не слетало. Однако совершенно не исключено, что там есть баги. Так что если кто все же надумает построить на этом девайс — поаккуратнее :)

Сам исходник лежит по ссылке. Предупреждаю — весь код в одном файле. Я очень не люблю структурность и раскидывание на много файлов :)

- **Архив с проектом** [1]

AVR. Учебный курс. Конечный автомат

Каждый кто пытался разбираться с конечными автоматами наверняка натыкался на всякие замудреные графы, какие то графики. Многие посчитав это слишком сложным плюнули и забили. А Зря!

С простейшим конечным автоматом каждый из нас сталкивался с самого детства — это механическая авторучка. Объект с единственной функцией «Нажатие кнопки», но в зависимости от очередности результат разный. Стержень то прячется, то вылазит.

Так и в нашем случае — конечный автомат это функция которая запоминает свое состояние и при следующем вызове делает свое черное дело исходя из прошлого опыта. Простой пример — мигалка (псевдокод):

```
1  ; Глобальные переменные
2  u08 Blink_State;
3
4  void Blink(void)
5  {
6  if (Blink_State == 1)
7          {
8          Led_On();
```

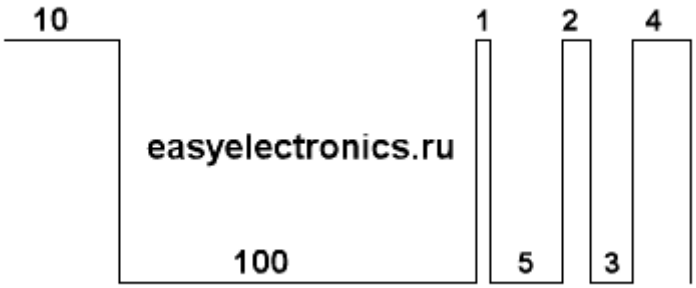
```
9           Blink_State = 0;
10          Return;
11          }
12 if (Blink_State == 0)
13          {
14          Led_Off();
15          Blink_State = 1;
16          Return;
17          }
18 }
```

Вызывая эту функцию подряд мы заставим диодик менять свое состояние при каждом вызове.

Но никто не мешает взять и сделать автомат куда сложней, на несколько десятков состояний. Никто не запрещает использовать вложенные конечные автоматы, никто не запрещает менять состояние автомата извне. В общем, мощнейший инструмент для построения быстрых и очень компактных алгоритмов.

Приведу другой, более сложный пример. Генерацию сигнала сложной формы.

Пусть у нас есть сигнал:



Цифрами указана задержка, в каких нибудь величинах. Это не суть важно.

Обычно народ не парится и лепит его по быдлокодерски, через delay:

```
1  Set_Pin();      // Вывод в 1
2  Delay(10);      // Задержка 10
3
4  Clr_Pin();      // Вывод в 0
5  Delay(100);     // Задержка в 100
6
7  Set_Pin();
8  Delay(1);
9
10 Clr_Pin();
11 Delay(5);
12
13 Set_Pin();
14 Delay(2);
15
16 Clr_Pin();
17 Delay(3);
18
19 Set_Pin;
20 Delay(4);
21
22 Clr_Pin();
```

Это дает минимальный код, но затыкает работу контроллера на весь период посылки. А если слать надо постоянно? Да еще дофига всего попутно делать? Экран обновлять, данные обрабатывать, в память писать… В таком случае у нас рулит RTOS, где на Delay происходит передача управления диспетчеру. Но если ОС нету? Вот тут то и идет в ход конечный автомат.

Проще всего тут применить таймер и его прерывание по переполнению. Таймер сам, в своем прерывании, будет загружать себя новыми значениями выдержки и щелкать дальше.

```
1  Timer_Overflow_Interrupt(void)
2  {
3  switch(TMR_State)                    // Обработчик прерывания по переполнению
4          {
5          case 0:
6                  {
7                  Clr_Pin();           // Вывод в 0
8                  TCNT = 255-100;      // Задержка в 100 (до переполнения)
9                  TMR_State = 1;       // Следующая стадия 1
10                 Break;               // Выход
11                 }
12
13         case 1:
14                 {
15                 Set_Pin();
16                 TCNT = 255-1;
17                 TMR_State = 2;
18                 Break;
19                 }
20
21         case 2:
22                 {
23                 Clr_Pin();
24                 TCNT = 255-5;
25                 TMR_State = 3;
26                 Break;
27                 }
28
29         case 3:
30                 {
31                 Set_Pin();
32                 TCNT = 255-2;
33                 TMR_State = 4;
34                 Break;
35                 }
36
37         case 4:
38                 {
39                 Clr_Pin();
40                 TCNT = 255-3;
41                 TMR_State = 5;
42                 Break;
43                 }
44
45         case 5:
46                 {
47                 Set_Pin();
48                 TCNT = 255-4;
49                 TMR_State = 6;
50                 Break;
51                 }
52
53         case 6:
54                 {
55                 Clr_Pin();
```

```
56                  Timer_OFF();            // Выключаем таймер. Работа окончена
57                  TMR_State = 0;          // Обнуляем состояние
58                  Break;
59                  }
60
61          default:        break;
62          }
63 }
```

А запускается эта байда простым пинком:

```
1 Set_Pin();
2 TCNT=255-10;              // Грузим таймер на выдержку 10 тиков
3 TMR_State = 0;            // Устанавливаем начальное положение
4 Timer_ON();               // Поехали!
5
6 ... // После чего можно заниматься чем угодно.
```

Когда автомат отработает — сама себя выключит. Можно еще какоенить событие сгенерировать, дабы основная программа поняла, что все уже готово. Хотя основная программа спокойно может палить переменную состояния автомата и все оттуда узнать сама.

Конечный автомат можно зациклить — скажем на стадии 6 сделать перенаправление на стадию 0, то получим генератор сигнала сложной формы. Причем он будет занимать минимум процессорного времени.

Если мы, например, захотим сделать десяток ШИМ сигналов? То что нам мешает повесить их на ОДИН таймер, главное отсортировать все скважности по возростанию, причем сортировать их вместе с ногами которыми нужно дрыгать. А потом прогнать по прерыванию таймера конечный автомат, да так чтобы он по стадиям передергивал ножки. Правда при изменении скважности любого из этих софтверных ШИМ каналов придется делать повторную сортировку всего массива. Разумеется больших скоростей мы на этом не получим. Таймер не может щелкать так быстро, да и математики там хватит. Но для многих задач, например, одновременное управление десятком сервомашинок этого более чем достаточно.

А если в том же прерывании таймера сделать выбор следующей стадии исходя не из тупой последовательности, а, скажем, на основе обрабатываемого байта, то мы получим программно-аппаратный генератор, к примеру, 1-Wire кода. Достаточно анализировать входной буфер и если там 1 — перебрасывать автомат в состояние соответствующее отработки генерации сигнала 1, а если 0, то в состояние генерящее выдержку нуля.

Более того, на автоматах можно полностью построить логику работы программы. От и до. Получается весьма торчково, пока не въедешь в это всей душой мозг сломать можно, но зато работает просто зверски и очень проста в отладке.

Не буду описывать это так как есть замечательный цикл статей Владимира Татарчевского, опубликованный в журнале Компоненты и Технологии. Настоятельно рекомендую к прочтению. Чтобы вам не шерстить по инету, я все части этой замечательной статьи выложил в **архив** [1]. Качайте и вкуривайте.

Особенно автоматный метод доставляет тем, что готовые автоматы вписываются как родные в ЛЮБУЮ почти архитектуру. У меня они и во флаговых автоматах работают на ура, и из диспетчера RTOS я их гоняю как родные. Красота!

AVR. Учебный Курс. Работа на прерываниях

Одним из серьезных достоинств контроллеров AVR является дикое количество прерываний. Фактически, каждое периферийное устройство имеет по вектору, а то и не по одному. Так что на прерываних можно замутить кучу параллельных процессов. Работа на прерываниях является одним из способов сделать псевдо многозадачную среду.

Идеально для передачи данных и обработки длительных процессов.

Для примера покажу буфферизированный вывод данных по USART на прерываниях.

В прошлых примерах был такой код:

```
1  // Отправка строки
2  void SendStr(char *string)
3  {
4  while (*string!='\0')
5          {
6          SendByte(*string);
7          string++;
8          }
9  }
10
11 // Отправка одного символа
12 void SendByte(char byte)
13 {
14 while(!(UCSRA & (1<<UDRE)));
15 UDR=byte;
16 }
```

Данный метод, очевидно, совершенно неэффективен. Дело в том, что у нас тут есть тупейшее ожидание события — поднятие флага готовности USART. А это зависит, в первую очередь, от скорости передачи данных. Например, на скорости 600 бод передача каких то 600 знаков будет длиться 9 секунд, блокируя работу всей программы, что ни в какие ворота не лезет.

Как быть?

Ну раз у нас отправка идет по аппаратному устройству, то большую часть времени мы впустую крутим цикл ожидания флага, хотя от процессора, собственно, тут ничего и не требуется и можно было бы заняться другими делами. Напрашивается мысль выбросить весь цикл ожидания в тело ядра/автомата/суперцикла и обрабатывать флаг на каждой итерации главного цикла/диспетчера. Но при этом у нас будет плавать время между байтами — ацтой!

Поэтому прерывания, пожалуй, будет единственным адекватным вариантов.

Итак, если брать в пример **USART** то у него есть три прерывания:

- RXT — прием байта. С этим понятно, мы его уже использовали
- TXC — завершение отправки
- UDRE — опустошение приемного буфера

Байты TXC и UDRE обычно вызывают путаницу. Поясню разницу.

Дело в том, что регистр передачи данных UDR в AVR на самом деле куда хитрей чем кажется, он двухэтажный. На первом ярусе, собственно UDR, а ниже находится конвейер сдвигового регистра. Первый байт, попавший в пустой регистр UDR тут же проваливается на конвейер, а UDR снова опустошается. После чего конвейер неторопливо, в соответствии с битрейтом, выплевывает данные в линию, а потом снова зажевывает байт из UDR. Поэтому, фактически, в UDR за короткое время влезает сразу два байта — первый тут же проваливается, а второй ждет.

Так вот,

- Флаг пустого регистра UDRE выставляется тогда, когда мы можем загнать байт в UDR,
- Флаг окончания передачи TXC появляется только тогда, когда у нас конвейер опустел, а новых данных в UDR нет.

Да, можно слать данные и по флагу TXC, но тогда у нас будет лишняя пауза между двумя разными байтами — время на опустошение буфера. Некошерно.

Вот как это можно сделать корректней.

Вначале выводим данные в массив, либо берем его из флеша — не важно. Для простоты запихну массив в ОЗУ. Код возьму из прошлой статьи:

```
#define buffer_MAX 16  // Длина текстового буффера
char buffer[buffer_MAX] = "0123456789ABCDEF";         // А вот и он сам
u08 buffer_index=0;                              // Текущий элемент буффера
```

Инициализация интерфейса выглядит стандартно:

```
//InitUSART
UBRRL = LO(bauddivider);
UBRRH = HI(bauddivider);
UCSRA = 0;
UCSRB = 1<<RXEN|1<<TXEN|0<<RXCIE|0<<TXCIE;
UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;

sei();          // Разрешаем прерывания.
```

Обратите внимание, что прерывания UDRE мы не разрешаем. Это делается потом. Иначе сразу же, на старте, контроллер **у**скачет на это прерывание, т.к. при пуске UDR пуст и мы получим черти что.

Отправка выглядит элементарно. Вначале пихаем первый байт нашего сообщения, не забыв выставить правильно индекс. А дальше разрешаем прерывание по UDRE оно само улетит куда надо, по прерываниям:

```
buffer_index=0;         // Сбрасываем индекс
UDR = buffer[0];                // Отправляем первый байт
UCSRB|=(1<<UDRIE);      // Разрешаем прерывание UDRE
```

Дальше можно затупить или делать вообще что угодно:

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

В течении нескольких тактов выскочит прерывание UDRE

Да, кстати, я один раз словил гадский баг который отловил только трассировкой ассемблерного листнига. У меня была такая последовательность:

```
UDR = X;
UCSRB|=(1<<UDRIE);
buffer_index = 1;
```

И вот тут почему то первым байтом шел мусор. А дальше все нормально. Причем если менять уровень оптимизации, то баг то вылезал то нет. Причиной такого поведения являлось то, что я то надеялся на то, что прерывание UDRE выскочит гораздо поздней чем я присвою индексу буфера нужное значение (buffer_index = 1;) Но индюк тоже думал, а по факту я
пихаю байт в UDR, он в тот момент естественно пуст и уже следующим тактом, на выполнении команды UCSRB|=(1<<UDRIE) данные проваливались в сдвиговый регистр, а UDR тотчас пустел и выставлял бит прерывания.
А дальше, в зависимости от оптимизации, этот бит успевал выставиться к моменту когда я выставлял верный номер индекса либо не успевал.
Проблема решилась перестановкой строк:

```
UDR = X;
buffer_index = 1;
UCSRB|=(1<<UDRIE);
```

Отсюда правило:
Готовь все необходимые данные ПЕРЕД разрешением прерываний.

```
// Прерывание по опустошению буффера УАПП
ISR (USART_UDRE_vect)
{
buffer_index ++;                        // Увеличиваем индекс

if(buffer_index == buffer_MAX)          // Вывели весь буффер?
        {
        UCSRB &=~(1<<UDRIE);    // Запрещаем прерывание по опустошению - передача
закончена
        }
        else
        {
        UDR = buffer[buffer_index];     // Берем данные из буффера.
        }
}
```

Все, автоматика! Каждый раз когда UDRE пустеет прерывание срабатывает и бросает туда новых дров. Когда же буфер пустеет и индекс достигает максимума, то мы просто запрещаем прерывание UDRE и успокаиваемся.

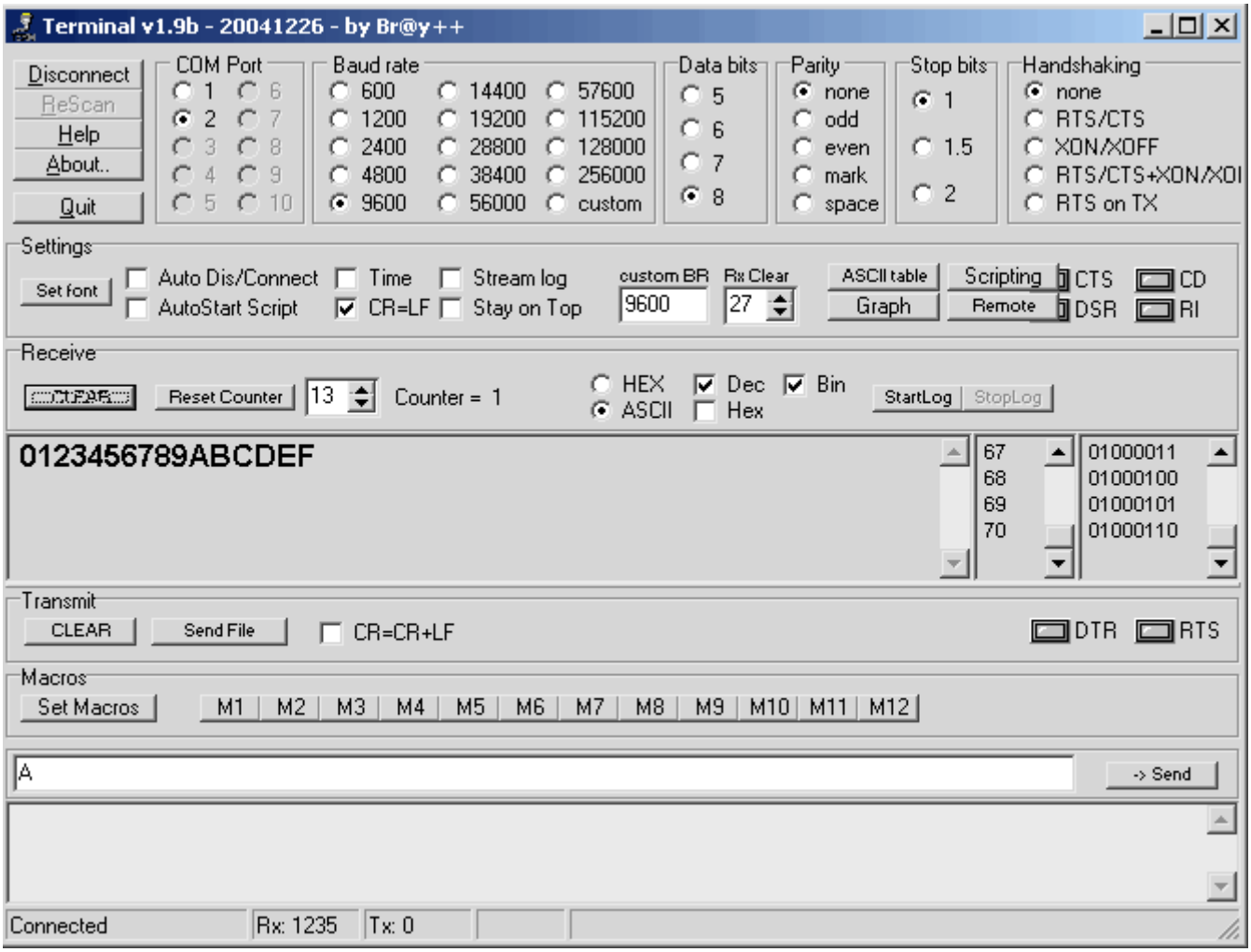
Осталось только дать понять головной программе, что мы отработали. Для этого и есть флаг TXC можно разрешить его прерывание и тогда он сбросится при обработке прерывания USART_TXC_vect, а в самом обработчике сделать заброс задачи на **д**испетчер или еще что нибудь умное. Либо периодически проверять главным циклом наличие флага TXC и вручную его стереть (записью единицы).

Вот полный код:

```
#define F_CPU 8000000L
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avrlibdefs.h>
#include <avrlibtypes.h>

#define buffer_MAX 16                           // Длина текстового буффера
char buffer[buffer_MAX] = "0123456789ABCDEF";         // А вот и он сам
u08 buffer_index=0;

//Прерывание по опустошению буффера УАПП
ISR (USART_UDRE_vect)
{
buffer_index++;                 // Увеличиваем индекс

if(buffer_index == buffer_MAX)          // Вывели весь буффер?
        {
        UCSRB &=~(1<<UDRIE);    // Запрещаем прерывание по опустошению - передача
закончена
        }
        else
        {
        UDR = buffer[buffer_index];     // Берем данные из буффера.
        }
}

int main(void)
{

#define baudrate 9600L
#define bauddivider (F_CPU/(16*baudrate)-1)
#define HI(x) ((x)>>8)
```

```
34 #define LO(x) ((x)& 0xFF)
35
36 //Init UART
37 UBRRL = LO(bauddivider);
38 UBRRH = HI(bauddivider);
39 UCSRA = 0;
40 UCSRB = 1<<RXEN|1<<TXEN|0<<RXCIE|0<<TXCIE;
41 UCSRC = 1<<URSEL|1<<UCSZ0|1<<UCSZ1;
42
43 //Это так, просто помигать.
44 #define LED1 4
45 #define LED_PORT PORTD
46 #define LED_DDR DDRD
47
48 LED_DDR = 1<<LED1;
49
50 sei();
51
52 buffer_index=0;         // Сбрасываем индекс
53 UDR = buffer[0];                // Отправляем первый байт
54 UCSRB|=(1<<UDRIE);      // Разрешаем прерывание UDRE
55
56 while(1)
57         {
58         LED_PORT=0<<LED1;
59         _delay_ms(1000);
60         LED_PORT=1<<LED1;
61         _delay_ms(1000);
62         }
   }
```

Если грузануть его в **Pinboard** [1], предварительно подключив USART к FT232 и законнектиться терминалкой, то будет мигать наш LED4, а в терминалку от стрелятся байты ASCII кодов нашей строки. В это же время будет неторопливо тикать наш цикл с мигалкой.



Ну и, как всегда, пример кода в архиве. [2]

Виртуальная машина и байт код

Часто требуется делать большие последовательности сложных операций — например полетное задание для робота. Да, можно запихать все это дело в основную программу, но вдруг что то пойдет не так как надо и алгоритм надо будет переделать — придется переделывать всю программу.

Тут на помощь придет виртуальная машина. Суть в чем — в памяти контроллера, в основную программу, занесены основные процедуры управления устройством. Если это робот, то это могут быть такие простые команды как «вперед», «назад», «повернуть» и так далее.

Дальше мы увязываем это все в таблицу переходов, где у каждой команды будет номер-смещение.

Потом нам нужен обработчик скриптов, который бы брал откуда нибудь, нашу последовательность действий — скрипт и преобразовывал это в вызовы реальных кусков кода -микрооперации.
Обработчик скриптов может быть той же самой задачей диспетчера, запущенной фоном. А откуда он будет брать данные неважно. Их вполне можно засасывать по usart или тащить из EEPROM памяти. А можно загнать на IIC память и получится сменный картридж :)

В простейшем случае, обработчик скрипта хватает байт-код из источника и сразу же заправляет его на конвейер, где микрооперация выполняется также как и обычная задача. А чтобы очередь диспетчера не переполнялась можно сделать так, чтобы новая партия байт-кода бралась только тогда, когда прожуется первая. Для этого достаточно, чтобы обработчик скрипта запускался по завершении каждой микрооперации.

В более продвинутой системе в байткоде могут быть и переменные. Тогда обработчик скрипта должен анализировать код, хотя бы по свитчу.

Например, есть у нас такая система команд:

- 0x00 выключить периферию
- 0x01 включить периферию
- 0x02,0xZZ поднять манипулятор на ZZ попугаев.
- 0×03,0xYY повернуть манипулятор на YY слонят.
- 0×04 открыть зажим
- 0x05 закрыть зажим

Тогда алгоритм хватания и переноса в нулевую позицию ящика стоящего на высоте в 10, под углом 15. Будет в источнике кода (ЕЕПРОМ или вообще что угодно) будет выглядеть примерно таким образом:

```
01, 02,10, 03,15, 04, 02,-10, 05, 02,10, 03,-15, 02,-10, 04, 00
```

Всего 17 байт на довольно сложное действие. Обрабатывать его можно будет примерно таким способом.

```
script_loader(void)
{
Byte_op=current_chain[i]       // Берем код команды

switch(Byte_op)
        {
        case 0x02:      // Если у ней есть аргументы
        case 0x03:      // То в зависимости от типа команды делаем переход
                {
                SetTask(Byte_op);               // Сразу грузим команду на конвейер
                i++;                            // Берем аргумент
                Param_REG = current_chain[i];   // Грузим его в регистр аргументов
                }                               // Откуда задаче будет его удобно взять
                break;
        default:                                // Если команда не имеет аргументов
                {
                SetTask(Byte_op)                // Просто бросаем ее на конвейер.
                Param_REG = 0;                  // А регистровый параметр зануляем
                }                               // Хотя это и не обязательно
        }
i++;            // Переводим указатель на следующую команду
}
```

Сейчас операнды байт-кода выполняются последовательно. НО! Никто не мешает сделать доп аргумент разрешающий выполнять их и одновременно (кооперативно, разумеется, но никто не мешает побить ту же микрооперацию на подзадачи которые можно перемешать в конвейере).
И тогда, например, команды подъема и поворота могут быть указаны следующим образом:

- 0x00 выключить периферию
- 0x01 включить периферию
- 0x02,0xZZ,0xMM поднять манипулятор на ZZ попугаев. Поставить MM код многозадачности
- 0×03,0xYY,0xMM повернуть манипулятор на YY слонят. Поставить MM код многозадачности
- 0×04 открыть зажим
- 0x05 закрыть зажим

Что такое код многозадачности? Ну это я только что, экспромтом, придумал :) Некий флаговый байт в памяти содержащий код, который показывает, что текущая задача может допускать параллельное выполнение других задач, а также свой собственный код.
При этом, управление обработчику скриптов отдается сразу же (если MM не нулевой) и следующая задача, забрасываемая в конвейер, проверит код многозадачности и если он не конфликтует с ней (понятно, что повернуть одновременно на разные углы нельзя или есть ограничения механического плана) — пихается на конвейер в нагрузку к первой.

Конечно, тут еще надо проверять кучу разных условий, думать как хранить коды многозадачности операций и вообще все несколько сложней, чем кажется на первый взгляд. Но общая концепция, думаю, ясна. А детали уже вылезут по ходу реализации.

Вот таким нехитрым образом, увязав диспетчер* с обработчиком скриптов мы получаем действующую виртуальную машину, со своим языком, своим байт кодом. Получаем полностью отвязанные от железа алгоритмы устройств.
А поскольку байт код может быть где угодно, хоть в ОЗУ, то это открывает нам широчайший простор для создания полиморфических и самообучающихся алгоритмов =).

Предлагаю коллективным разумом на форуме [1] сгенерировать ROBO-API на подобном байткоде. По крайней мере сформулировать список команд, синтаксис инструкций. А тут, в комментах, обсуждения самого подхода.

*) Диспетчер рассмотренный в прошлых статьях Архитектуры Программ курса AVR для этих целей не очень годится, т.к. конвейер там основан на указателях, а не на таблице переходов (хотя никто не мешате сварганить там и таблицу переходов, прям в задаче скрипт лоадера). А вот диспетчер из цикла статей по ассемблерной RTOS вполне на эту роль пойдет. Ну и кто то его на Си портировал уже.

Пример виртуальной машины

Как то раз я описывал концепцию создания языка программирования для устройства [1]. Который бы позволил запихать сложнейший алгоритм или последовательность действий в виде компактного скрипта.

Простой пример для чего это нужно — фрезерный станок с ЧПУ. И надо на нем выточить голову Сократа из цельного куска металла. Задача, на самом деле, не шибко сложная.

Но попробуйте написать прошивку, двигающую резцом, в виде классического конечного автомата — двигающую резцом в зависимости от условий или состояний. Да вы сдохнете раньше чем это сможете сделать.

Другое дело если разбить программу на элементарные операции, вроде «Резец вверх», «резец вниз», «шаг на n мм», а прошивке скормить последовательность этих микроопераций в виде байт-кода или текстового скрипта. Как все серьезно упрощается. Да и попутно можно нашинковать Платона с Гераклом, было бы желание, да образец для копирования.

Т.е. у нас появился свой язык устройства, полностью отвязанный от аппаратной реализации и оперирующий только органами устройства. И вот тут, главное, не впасть в высокоуровневую прелесть и не начать изобретать универсального интерпретируемого языка аля JAVA для микроконтроллеров. В условиях ограниченных ресурсов это полный бред.

Наоборот, этот язык нужно максимально упрощать за счет усложнения процедур машины, затачивая их строго под текущую задачу, а не делая нечто универсальное. Тогда можно выжать максимум из контроллера.

Ну, хватит воды, приведу пример того, что у меня получилось за вечерок курения в код. Код рабочий, но я там ничего не оптимизировал. Так, накидал чтоб работало, до ума доведете сами. Сделано все на базе ядра диспетчера. Я его **уже описывал** [2], поэтому работу его функций пояснять не буду

Итак, начну разматывать с самого верха. Есть у нас программа:

- FWD
- DLY,T(1000)
- STP
- DLY,T(1000)
- BCK
- DLY,T(2000)
- STP
- OFF

Очень похожа на ассемблер, да она по сути дела им и является — это ассемблер устройства. В данном случае некой самобеглой тележки. Все на примитивнейшем уровне, только чтобы показать идею.

- FWD — ехать вперед.
- DLY — задержка, в скобочках время в мс
- STP — стоп
- BCK — назад
- OFF — выключение.

Простенькая такая программка истинного партийца — шаг вперед, два шага назад. =)))

Программку я оформил в файле **VM_PROG.h** [3], чтобы не путаться где у нас что я буду звать этот скрипт **надпрограммой**

```
1  #define OFF 0
2  #define BCK 1
3  #define FWD 2
4  #define DLY 3
5  #define STP 4
6  #define T(X)  ((X)& 0xFF),((X)>>8)
7
8  static u08 VM_PGM[]=
9
10 {
11 FWD,              //0
12 DLY,T(1000),     //1,2,3
13 STP,              //4
14 DLY,T(1000),     //5,6,7
15 BCK,              //8
16 DLY,T(2000),     //9,10,11
17 STP,              //12
18 OFF               //13
19 };
```

Тут все просто. Вначале через дефайны мы каждой мнемонике нашего ассемблера присваиваем номер команды (это важно!). А потом забиваем их в стандартный массив во флеше. Массив занял у нас 13 байт. Обрати внимание на то, что команды переменной длины. Т.е. есть простые — однобайтные, а есть двубайтные, например, задержка. Там байт на команду и два байта на выдержку.

Каждой команде присвоена своя процедура, содержимое процедур не важно особо, они для примера. Тут у меня «вперед» это зажигание одного светодиода на Pinboard [4], а «назад» другого. Команды эти лежат в файле **VM.c** [5], а хидеры в **VM.h** [6] :

```
1 void VM_Back(void)
2 {
3 LED_PORT  ^=1<<LED3;           // Зажигаем диод
4 VM_PC[0]++;                    // Выбор следующей задачи VM
5
6 SetTask(VM);                   // Вброс диспетчера виртуальной
7                                // машины на конвейер диспетчера RTOS
8 }
```

Как видишь, тут мы что то делаем, а потом увеличиваем счетчик виртуальной машины. Заставляя выбирать ее следующую команду из нашей надпрограммы. Дальше обработчик виртуальной машины закидывается на конвейер ядра RTOS. Впрочем тут нет разницы каким образом организована главная логика программы. Это может быть и конечный автомат или же флаговый автомат. Остальные подзадачи аналогичны.

```
1 void VM_Stop(void)
2 {
3 LED_PORT  ^=1<<LED3;
4 VM_PC[0]++;
5
6 SetTask(VM);
7 }
```

```
1           void VM_Delay(void)
2           {
3           u16 delay;                              // Двубайтная переменная
4           u08 *OneByte;                           // Указатель на один байт
5
6           OneByte = (u08 *)&delay;                // Берем адрес переменной
7
8           *OneByte = VM_PGM[++VM_PC[0]]; // По байтикам собираем слово
9           OneByte++;
10          *OneByte = VM_PGM[++VM_PC[0]];
11
12          VM_PC[0]++;                             // Прощелкиваем счетчиком надпрограммы
13
14          SetTimerTask(VM,delay);                 // Запуск задачи VM через диспетчер
15          таймеров.
            }
```

Программа обработчика оператора задержки чуток сложней. Она трехбайтная. И поэтому забирает из массива надпрограммы байты временной выдержки, увеличивая счетчик на три. А дальше также запускает обработчик VM, но уже с выдержкой.

```
1 void VM_OFF(void)
2 {
3 InitVM();
4 }
```

Тут просто идет переинициализация VM

Сами обработчики операторов надпрограммы уложены в таблицу переходов, которая размещена во флеше.

```
1 //VM Task Table
2 static TPTR VM_FUNC[] PROGMEM =
3    {
4    &VM_OFF,            //0
5    &VM_Back,           //1
6    &VM_Forward,        //2
7    &VM_Delay,          //3
8    &VM_Stop            //4
9    };
```

И вот тут важная деталь! Расположение адресов в таблице переходов ТОЧНО соответствует коду операции. Т.е. по нашей надпрограммной системе команд у команды OFF код 0, и ее адрес находится в нулевой ячейке массива VM_FUNC[].

И дальше все получается очень и очень просто! Мы тупо берем байты из которых состоит надпрограмма и по таблице переходов перебрасываемся на нужный обработчик. Переброску осуществляет диспетчер RTOS. Поэтому процедура обработки диспетчера виртуальной машины всего из одной строчки:

```
1 void VM(void)     // Виртуальная машина
2 {
3 SetTask((void*)pgm_read_word_near(VM_FUNC+VM_PGM[VM_PC[0]]));
4 }
```

То есть тут только вызов **SetTask**. Куда мы считываем ему адрес из памяти программ, из массива VM_FUNC, по смещению из массива VM_PGM, а само смещение берем из массива VM_PC. Вот такая вот Кащеева смерть *(Кащей, кстати, знаю что читаешь — заходи бухать. Давно тебя не видел)*.

А что за массив VM_PC? А это программный счетчик нашей виртуальной машины. Переменная в нем дает значение смещения по массиву с надпрограммой. Для выборки следующей выполняемой команды надпрограммы. Т.е. исходя из значения в VM_PC мы берем значение следующей команды. Поэтому то мы в каждой задаче и увеличивали его значение на 1, а если шли данные, то на 1+величину этих данных (в нашем случае на 1+2).

Заботу о программном счетчике приходится решать нам, в обработчиках операторов надпрограммы. А как ты хотел? Реальный процессор работает точно также :)))) А мы сделали процессор в процессоре.

Меняя значение в VM_PC мы можем делать переходы на нужный оператор, а добавив в операнды обработчик условия получим в нашей надпрограмме ветвления и циклы. Если нужно, конечно. Главное правильно вычислять адрес, ведь у нас команды могут быть и многобайтными, а значит легко выполнить данные вместо кода.

Остается один только вопрос, а нафига VM_PC массив? Ведь хватит и одного байта. Хватит, ага, но кто сказал, что у нас может выполняться только одна надпрограмма? ;))))

Почему бы не запустить несколько параллельных процессов, каждая со своим VM_PC?

Надо только в обработчики операторов надпрограммы передавать номер обрабатываемой ветви, да следить чтобы конвейер ядра RTOS не сорвало. Плюс придумать что то с очередью таймеров. Т.к. сейчас DELAY прописывается в единственном числе под каждую задачу. Так что у нас не может быть в очереди таймеров, скажем, два FWD на ожидании. Также надо будет решить over 9000 проблем связанных с блокировками общих ресурсов и прочим загоном многозадачных систем. Но все эти проблемы давно уже описаны и известны, так что курить книжки Таненбаума [7] и дорогу осилит идущий.

Запускается же виртуальная машина просто:

Вначале инициализация программных счетчиков, чтобы поехать точно с начала

```
1 void InitVM(void)
2 {
3 u08 i;
4
5 for(i=0;i!=10;i++)
6    {
7    VM_PC[i]=0;
8    }
9 }
```

А дальше запускаем ее как обычную задачу из под диспетчера:

```
1  int main(void)
2  {
3  InitAll();              // Инициализируем периферию
4  InitRTOS();             // Инициализируем ядро
5  InitVM();
6  RunRTOS();              // Старт ядра.
7
8  // Запуск фоновых задач.
9  SetTask(VM);
10
11 while(1)                // Главный цикл диспетчера
12 {
13 wdt_reset();           // Сброс собачьего таймера
14 TaskManager();         // Вызов диспетчера
15 }
16
17 return 0;
18 }
```

Ну и, напоследок, как обычно, собранный проект для WinAVR+AVRStudio с этим примером

WM-GCC-RTOS.ZIP [8]

З.Ы.
На сайт теперь прикручена кодопомойка с подсветкой синтаксиса. Цветами я еще поиграюсь, но уже работает. Теперь не будет проблемы с публикацией длинных кусков кода в комментарии. Достаточно разместить ее в

кодопомойке, а в коммент вставить ссылку. Ссылка на кодопомойку встроена теперь в поле ответа на коммент (открывается в новом окне).

Кодопомойка [9]

Работает только для зарегистрированных участников.

3.3.Ы.
В **Карте Сообщества** [10] был добит зверский баг из-за которого многие точки теряли координаты и топились в Атлантическом Океане, где то в районе Африки. Глюк мы подчистили, а также зачистили базу с битыми точками. Битых точек было вагон, поэтому если кто ставил свои отметки — проверьте их наличие, возможно их уже там нет. Надо переставить. Ну и народу там понадобавлялось порядочно. Сама же карта теперь торчит в виде баннера в виде… хм, карты.

3.3.3.Ы.
У меня тут, ВНЕЗАПНО, родственники, ТЫСЯЧИ ИХ! Плюс еще дела мелкие, но по тому же сценарию. Так что не теряйте. Я не забил, просто очень занят :(

Использование интерфейса USI в режиме мастера TWI

Возникла необходимость использовать EEPROM совместно с контроллером ATTiny44. Соответственно выбор пал на AT24C64, работающую по интерфейсу I^2C (TWI по атмеловской терминологии). Порыл в документации и в инете — с виду вроде все просто, но при реализации алгоритма несколько раз возникали вилы нигде толком не обозначенные, поэтому решил написать статейку.

Я не мега-гуру, поэтому если будут ошибки или недочеты — надеюсь спецы в комментах поправят. Код буду приводить на Си в простом быдло-кодерском исполнении, так как пока нет нужды заморачивать более сложные варианты. Теперь обо всем по порядку…

Примечание: для понимания материала статьи желательно представлять работу TWI интерфейса и микросхем serial eeprom. Прочитать об этом можно в статьях DI HALT'а про IIC интерфейс [1] и, например, в статье про часы [2] (где-то еще попадалась статья про работу конкретно с serial eeprom, но не нашел…)

Так же я люблю всякие команды типа cbi и sbi, поэтому для корректной работы примеров необходимо вставить в начало программы следующую строчку:

```
1 #include<compat/deprecated.h>
```

и еще вставить строчку в разделе определения типов :

```
1 typedef unsigned char u08;
```

USI
Интерфейс USI присутствует во многих контроллерах семейства Tiny и части семейства Mega и представляет собой полуфабрикат-заготовку из сдвигового регистра, 4х-битного счетчика и набора регистров инициализации с прерываниями. Может работать как в 3х-проводном режиме (как SPI) так и в 2х-проводном режиме (TWI или I2c), так же у него есть нестандартные применения для реализации UARTов и прочего. Я, например, успешно пихал через него данные в цепочку 74hc595 с 7ми-сегментниками :)

Полностью расписывать весь интерфейс я не буду, это есть в даташитах и в литературе (например А.В. Евстифеев. Микроконтроллеры AVR семейства Tiny.)
Для управления интерфейсом USI у нас есть следующие регистры:

- **USICR** — регистр управления
- **USISR** — регистр состояния
- **USIDR** — регистр данных
- **USIBR** — буферный регистр только для чтения (отсутствует в некоторых моделях AVR, типа tiny2313, смотрите даташит)

Регистр данных **USIDR** представляет собой стандартный регистр ввода-вывода, никаких особенностей кроме той, что его содержимое автоматом сдвигается и вываливает значение старшего бита на соответствующую ногу контроллера.
Буферный регистр **USIBR** — копия **USIDR** только для чтения. В него автоматом копируется значение из USIDR после каждой операции сдвига.
Регистр состояния **USISR**— регистр ввода-вывода, который в старших 4х разрядах содержит флаги прерываний по обнаружению состояний СТАРТ, СТОП, по переполнению 4х-битного счетчика и по обнаружению коллизий при выводе данных, а в младших 4х разрядах содержит 4х-битный счетчик.

ВАЖНО!!!
Обратите внимание: ЧТОБЫ СБРОСИТЬ программно какой-либо флаг прерывания — надо УСТАНОВИТЬ соответствующий бит регистра в 1! Это справедливо почти для всех флагов любой периферии AVR

Регистр управления **USICR** я распишу немного подробней, т.к. нам его нужно правильно инициализировать:

- 7ой бит **USISIE** — разрешение прерывания по обнаружению состояния СТАРТ
- 6ой бит **USIOIE** — разрешение прерывания при переполнении счетчика
- 5ый бит **USIWM1** и 4ый бит USIWM0 — режим работы модуля USI
- 3ий бит **USICS1** и 2ой бит USICS0 – выбор источника тактового сигнала
- 1ый бит **USICLK** — строб тактового сигнала
- 0ой бит **USITC** — переключение состояния вывода тактового сигнала

Выглядит немного непонятно, но более подробное описание всех значений есть в документации а далее по тексту я постараюсь объяснить значения тех или иных битов чтоб было проще и наглядней…

Далее я буду рассматривать только 2х-проводной режим работы интерфейса.
А еще у нас в режиме TWI есть две ноги обозначенные как SCL и SDA (для Tiny44 это выводы PA4 и PA6 соответственно).
Начнем с обозначения этих ног. Я обычно делаю отдельный файл с описанием периферии и называю его controller.h :

```
1 // USI pin association
2 #define sclport         PORTA
3 #define sclportd        DDRA
4 #define sdaport         PORTA
5 #define sdaportd        DDRA
6 #define scl             4
7 #define sda             6
8 #define sclportinp      PINA
9 #define sdaportinp      PINA
```

ВАЖНО!
В процессе работы с интерфейсом выводы scl и sda должны быть сконфигурированы на выход (в режиме работы от модуля USI они подключаются к выходному буферу интерфейса при установленном в 1 своем бите DDRx)!!! Но в режиме приема данных по линии sda данный вывод надо конфигурировать на вход!!!

Небольшое отступление
Общая схема у нас представляет собой Tiny44, затактованную от кварца частотой 7.3728 МГц с подключенным к линиям scl и sda чипом памяти at24c64. Линии scl и sda подтянуты к питанию через внешние резисторы 10k. Собственно, подключение стандартное для IIC интерфейса.

Инициализация интерфейса
Пример сделан на базе атмеловского апнота AVR310 [3]Что-то оставил как есть, кое что изменил, но внешнее сходство осталось сильным.

Для начала определим разного рода задержки т.к. скорость TWI все-таки не очень высокая.

```
1 #define USI_DELAY 20  // собственно сама задержка, подобранная экспериментально.
```

Задержку я подобрал экспериментально, не заморачиваясь стандартными скоростями I^2C. Мне важна была стабильность приема и передачи информации. Стандартную скорость вполне можно подобрать при желании, хоть в режиме standart, хоть в режиме fast.

Теперь сделаем предустановки значений четырех-битного счетчика. Этот счетчик используется для отсчета переданных бит данных. У нас пакет данных представляет собой 8 бит данных и один бит подтверждения. В итоге 9 бит.

```
#define USISR_8BIT 0xF0 // значение 4х битного счетчика для передачи 8 бит пакета информации
#define USISR_1BIT 0xFE  // значение счетчика для передачи 9го бита пакета информации
```

Сама процедура инициализации интерфейса взята с AVR310:

```
void init_usi_i2c_master(void)
{

// init USI
        sbi(sclportd,scl);
        sbi(sdaportd,sda);                                              // Preload dataregister with "released level" data.
        USICR    =  (0<<USISIE)|(0<<USIOIE)|                            // Disable Interrupts.
                    (1<<USIWM1)|(0<<USIWM0)|                            // Set USI in Two-wire mode.
                    (1<<USICS1)|(0<<USICS0)|(1<<USICLK)|                // Software stobe as counter clock source
                    (0<<USITC);
        USISR    =  (1<<USISIF)|(1<<USIOIF)|(1<<USIPF)|(1<<USIDC)|      // Clear flags,
                    (0x0<<USICNT0);
        sbi(sdaport,sda);
        sbi(sclport,scl);
        return;
}
```

В нашем случае прерывания нам не нужны, поэтому бит USISIE в 0 и USIOIE тоже в 0.
Далее переводим интерфейс в 2хпроводной режим без удержания линии scl, следовательно бит USIWM1 в 1, а бит USIWM0 в 0.

Если надо работать на прерываниях или в режиме слэйва, то удобней использовать режим, когда оба этих бита установлены в 1. Тогда при уходе в прерывание контроллер автоматически прижмет линию scl в 0 и устройства на шине будут ждать завершения обработки принятых или переданных вами данных.

Далее выставляем очень интересную комбинацию бит:

- USICS1 в 1
- USICS0 в 0
- USICLK в 1

Все возможные комбинации этих битов описаны подробно в даташите. Наша комбинация выбирает источником тактового сигнала для 4х-битного счетчика нашу программу, а еще точнее говорит о том, что линия SCL изменять свое состояние на противоположное при записи в бит USITC единицы и четырехбитный счетчик будет инкрементироваться, а регистр данных будет сдвигаться.

То есть для того чтобы выпнуть значение из регистра данных (и ли принять пришедшее в режиме мастера) нам надо 16 раз дернуть бит USITC в единицу! Фактически это ручной режим.

Ну и пока ставим USITC в 0, чтобы интерфейс пока ничем не дергал.

Также нельзя забывать о конфигурации ног SCL и SDA на выход. С этого момента старший бит регистра USIDR присутствует в виде уровня на ноге SDA! Обе линии находятся в исходном состоянии (в единице).

Теперь опишем процедуры для задания состояний СТАРТ и СТОП.

- Состояние СТАРТ — это изменение уровня SDA с высокого на низкий при высоком уровне линии SCL
- Состояние СТОП — это изменение уровня SDA с низкого на высокий при высоком уровне линии SCL

Я перестраховался с этими состояниями, поэтому код несколько избыточен и в тоже время прост как пробка и нагляден:

```
void i2c_start(void)
{ // Генерируем состояние Старт (или ПовСтарт)

        sbi(sdaport,sda);                       // на всякий случай выставляем в исходное
состояние sda
        sbi(sclport,scl);                       // тоже с SCL
        cbi(sclportd,scl);                      // ВАЖНО!!! отключаем SCL от выходного
буфера интерфейса
        USISR = (1<<USISIF)|(1<<USIOIF)|(1<<USIPF)|(1<<USIDC)|(0x0<<USICNT0);
        //сбрасываем USISR
        cbi(sdaport,sda);                       // переводим  SDA в 0 пока SCL в 1
        dummyloop(USI_DELAY);                   // тупим нашу задержку
        sbi(sclportd,scl);                      // ВАЖНО!!! подключаем SCL обратно к
выходному буферу интерфейса
        cbi(sclport,scl);                       // переводим SCL в 0
        sbi(sdaport,sda);                       // освобождаем линию SDA для последующей
передачи/приема данных
        dummyloop(USI_DELAY);                   // еще раз тупим задержку
        return();
}
void i2c_stop(void) {// Генерируем состояние Стоп

        cbi(sclport,scl);                        // необязательная подготовка
        dummyloop(USI_DELAY);
        cbi(sdaport,sda);
        dummyloop(USI_DELAY);
        sbi(sclport,scl);                       // перевод SCL в 1
        dummyloop(USI_DELAY);
        sbi(sdaport,sda);                       // перевод  SDA в 1
        dummyloop(USI_DELAY);
        USISR|=(1<<USIPF);                      //сброс флага детекции состояния Стоп в
USISR
        return ();
}
```

Перестраховка в коде заключается в том, что мои процедуры сгенерируют состояния Старт и Стоп независимо от того в каком фактическом состоянии находилась шина. То есть я сразу говорю что я тут один Мастер и других мастеров на шине быть не может. Потому все равняйсь, смирно, хватит фигню в шину пихать… Если у вас несколько мастеров — так однозначно делать не следует

Процедуру i2c_stop можно ускорить/сократить убрав лишние дерганья ногами и задержки.

ВАЖНО!
Обратите особое внимание на строчки выделенные в коде пометкой «Важно». В апнотах этих строчек нет! Если вы не сделаете простую операцию по отключению ноги scl от выходного буфера — то будете долго ломать голову, почему у вас ни чего не работает, какой фиговый интерфейс, и какие лузеры его разрабатывали и т.д. Я сам неделю ломал голову (с аналоговым осцилографом). На цифровике — можно быстро дотумкать запустив аппаратный TWI на той-же Меге и сравнив сигналы, но нету у меня цифры…

А собака порылась вот где: в мануалах, даташитах и литературе вскользь упоминается, что у контроллера есть такая «схема детекции состояния Старт», а в описании битов USIWM1..0 для двухпроводного режима есть фраза, что «драйвер линии SCL формирует на ней низкий уровень либо по сигналу детектора состояния Старт, либо если соответствующий линии бит сброшен в 0».

Поэтому при подключенном драйвере линии SCL, как только вы скажете cbi(sdaport,sda) контроллер моментально сдетектит «Старт» и сразу просадит в 0 SCL.

Если у вас в качестве ведомого будет такая же AVR-ка с USI-интерфейсом — то все прокатит и заработает, что нам успешно демонстрирует связка атмеловских апнотов AVR310-AVR311. А вот если в качестве ведомого у вас EEPROM-ка или тот же таймер 8583 — то он вас просто не поймет, т.к. между переходом sda и scl в 0 должна быть пауза.

Ну и, соответственно, после того как мы сделали свое черное дело, надо сбросить все флаги и счетчик в регистре состояния и вернуть на место драйвер шины SCL.

Процедура задержки. Я использую следующий вариант:

```
1  #define nop() asm volatile ("nop")
2
3  void dummyloop(unsigned int);
4
5  void dummyloop(unsigned int timetoloop)
6  {
7          while (timetoloop>0)
8          {
9          nop();
10         timetoloop--;
11         }
12 }
```

Скорей всего по быдлокодерски, но работает и не режется оптимизатором.

Дальше пишем процедуру приема/передачи:

```
   u08 usi_i2c_master_transfer(u08 tmp_sr) {

1          USISR = tmp_sr;                                    // Загружаем USISR нужным нам
2  значением
3          tmp_sr= (0<<USISIE)|(0<<USIOIE)|
4                  (1<<USIWM1)|(0<<USIWM0)|
5                  (1<<USICS1)|(0<<USICS0)|(1<<USICLK)|   // задаем битовую маску для USICR
6                  (1<<USITC);                            // самый важный бит. Ради него и
7  сделана эта переменная
8
9          do
10         {
11         dummyloop(USI_DELAY);                  // курим бамбук
12         USICR=tmp_sr;                          // запинываем значение в USICR, интерфейс
13 работает
14         while (bit_is_clear(sclportinp,scl)); // проверяем, свободна-ли линия
15         dummyloop(USI_DELAY);                  // снова курим бамбук
16         USICR=tmp_sr;                          // еще раз запинываем USICR
17         }
18         while (!(USISR&(1<<USIOIF)));          // повторяем предыдущие операции до
19 переполнения счетчика
20
21         dummyloop(USI_DELAY);                  // тупим в цикле
22
23         tmp_sr=USIDR;                          // сохраняем принятые данные
24         USIDR=0xff;                            // освобождаем линию sda
25         sbi(sdaportd,sda);                     //ВАЖНО!!! восстанавливаем  подключение SDA
26 к выходному буферу интерфейса
           return (tmp_sr);                       // Возвращаем принятые данные

   }
```

Данная процедура сдернута мной с апнота и почти без изменений. Примечательна она тем, что осуществляет как чтение так и передачу данных.
Работает она так:

ВАЖНО!
Данная процедура предполагает, что регистр USIDR уже загружен нужными данными для передачи!

Мы получаем от программы значение регистра USISR, с предустановленным счетчиком. Так как наш минимальный пакет данных составляет 8 бит данных в одну сторону и 1 бит в противоположную (смотрите логику работы TWI-интерфейса), то мы должны послать ведомому 8 бит и принять один (или же принять 8 бит послать один если передача идет в другую сторону).
Раз у нас есть аппаратный счетчик, да еще и с флагами, значит не надо городить кучу переменных в циклах и их проверки. Достаточно предустановить счетчик и гонять цикл до его переполнения.
Именно для этого мы определили константы USISR_8BIT и USISR_1BIT, которые задают значения счетчика для обработки 8бит данных и 1 бита данных, а заодно еще и сбрасывали ненужные и нужные флаги состояния.

Далее мы:

- Толкаем в бит USITC единичку, линия scl начинает менять свое состояние из 0 в 1, счетчик тикает увеличиваясь на 1 и приемник считывает первый переданный нами бит.
- Проверяем свободна-ли линия — если ведомое ведомое устройство не успевает — оно прижмет SCL в 0. В принципе это должно работает, но могут возникнуть вилы. У меня не было возможности расковырять данный момент досконально, а ведомый вроде всегда успевал.
- Делаем задержку.
- Еще раз толкаем в бит USITC единичку. Линия SCL при этом у нас возвращается в 0, регистр USIDR сдвигается на один бит, выпинывая в линию SDA новое значение, а счетчик увеличивается еще на 1.

Таким образом, мы на один проход цикла получаем полный такт передачи 1 бита по шине. Счетчик при этом тикает 2 раза на один проход.
Далее

- Повторяем все до переполнения.
- Делаем задержку.
- Сохраняем полученные данные.
- Возвращаем линию sda в нормальное состояние (1), для чего запинываем в USIDR любое значение с установленным 7мым битом.
- Восстанавливаем подключение линии SDA к выходному драйверу. Дело в том, что при передачи данных от мастера к ведомому драйвер у нас всегда подключен, чтобы линия SDA всегда соответствовала значению 7го бита USIDR, а если мы принимаем данные от ведомого, то линия SDA должна быть сконфигурирована как вход. Если этого не сделать — то у меня мастер не мог получить бит подтверждения и другие данные от ведомого. За направлением приема-передачи следит процедура отвечающая за протокол обмена, а в нашей процедуре мы просто восстанавливаем подключение, если оно было сброшено.

Это была процедура непосредственно обмена по интерфейсу, а теперь рассмотрим 2 процедуры более высокого уровня, которые уже реализуют часть протокола обмена данными:

Передача N-ного количества байт от мастера к ведомому

```
1  u08 i2c_master_send(u08 *data, u08 data_size)
2  {
3
4          if (bit_is_set(sclportinp,scl)) return(1);              // проверка, если старт не
5  прошел — выход с состоянием (1)
6
7          do
8          {
9          USIDR=(*(data++));                                    // загружаем очередной байт
10 данных
11         usi_i2c_master_transfer(USISR_8BIT);                  // посылаем 8 бит
12         cbi(sdaportd,sda);                                          // переключаемся на
13 прием
14         if ((usi_i2c_master_transfer(USISR_1BIT))&0x01) return(2);    // если нет
```

```
15 подтверждения - выход (2)
16         data_size--;                                                    // уменьшаем счетчик
17 данных
          }
          while (data_size>0);                                    // и так пока все не
   передадим

          return(0);                                              // успешная передача —
   выход с состоянием (0)
   }
```

Прием N-ного количества байт от ведомого к мастеру

```
   u08 i2c_master_read(u08 *data, u08 data_size)
   {
1          if (bit_is_set(sclportinp,scl)) return(1);             // проверка, если старт не
2  прошел — выход с состоянием (1)
3          do
4          {
5          cbi(sdaportd,sda);                                      // переключаемся на прием
6          *(data++)=usi_i2c_master_transfer(USISR_8BIT);                  // принимаем 8 бит
7
8          if (data_size==1)
9                  {
10                 USIDR=0xFF;                             //если последний байт — передаем
11 NACK
12                 }
13         else
14                 {
15                 USIDR=0x00;                             // если не последний — передаем
16 подтверждение ACK
17                 }
18
19         usi_i2c_master_transfer(USISR_1BIT);            // собственно передача ACK/NACK
20         data_size--;                                    // уменьшаем счетчик данных
21         }
22         while (data_size>0);                            // и так пока все не примем
23         return(0);                                      // успешный прием данных, выход с
24 состоянием (0)

   }
```

Процедуры простые и особых пояснений вроде не требуют.

Вот собственно и вся база для организации обмена по USI в режиме мастера.

Дальше уже реализация самого верхнего уровня протокола, зависящего от устройства с которым мы будем общаться.

К статье приложено несколько файлов
Они в архиве **usi.zip** [4]

- usi_i2c.c Исходники рассмотренные здесь
- i2c_memory.c , в котором набор процедур для работы с eeprom. Процедуры очень простые, но если возникнут вопросы — пишите. Расписывать их подробно — еще такая же статья получится. За раз сложно для понимания… Но если ногами не запинают напишу в виде продолжения…

AVR. Учебный Курс. Использование AVR TWI для работы с шиной IIC (i2c)

Про шину IIC я писал уже неоднократно. Вначале было описание протокола [1], потом пример работы в лоб [2], а недавно камрад Ultrin выложил пример работы [3] с i2c на базе блока USI. Да и в интернете полно статей по использованию этой шины в своих целях. Одно плохо — все они какие то однобокие. В подавляющем большинстве случаев используется конфигурация «Контроллер-Master & EEPROM-Slave». Да еще и на программном мастере. И ни разу я не встречал материала, чтобы кто то сделал Контроллер-Slave или описал многомастерную систему, когда несколько контроллеров сидят на шине и решают арбитражем конфликты передачи. Пустоту пора заполнять, решил я и завязал узелок на память… Да только веревочку пролюбил :)

Обещаного три года ждут, вот я таки пересилил лень, выкроил время и сообразил полноценную библиотеку для работы с аппаратным модулем TWI, встроенным во все контроллеры серии MegaAVR. Давно грозился.

Кошмар на крыльях ночи
Во-первых, я сразу же отказался от концепции тупого последовательного кода. Когда у нас есть некоторая функция SendByte(Address,Byte) которая шлет данные по шине, а потом возвращает 1 или 0 в зависимости от успешности или неуспешности операции. Метод прост, дубов, но медленный. Т.е. пока мы байт не пошлем мы не узнаем ушло ли оно, а значит будем вынуждены тупить и ждать. Да, шина i2c может быть очень быстрой. До 100кбит ЕМНИП, но даже это время, а я все же за высокоскоростное выполнение кода, без тормозных выдержек. Наш выбор — диспетчеризация и работа на прерываниях.

Суть в том, что мы подготавливаем данные которые нужно отослать. Дальше запускаем аппаратуру передачи и возвращаемся к своим делам. А зверский конечный автомат, что висит на прерывании TWI передатчика сам передает данные, отвлекая основную программу только тогда, когда нужен какой-либо экшн (например сунуть очередной байт в буфер передачи). Когда же все будет передано, то он генерит событие, которое оповещает головную программу, что мол задание выполнено.
Как? Ну это уже от конкретной реализации событий программы зависит. Может флажок выставить или байт состояния конечного автомата подправить, а может и задачу на конвейер диспетчера набросить или Event в почтовый ящик задачи скинуть. Если юзается RTOS.

Что у нас есть?
А у TWI всего один вектор прерываний (в отличии от USART, где на каждое событие по прерыванию). Впрочем, это не страшно. Т.к. к нему в нагрузку идет регистр TWSR в котором записывается код причины по которой мы оказались в прерывании. Т.е. у нас, без лишних проблем, образуется конечный автомат.

Вот перечень кодов выдаваемых этим регистром
Коды состояний для режима Master

- **0×00 Bus Fail** Автобус сломался… эээ в смысле аппаратная ошибка шины. Например, внезапный старт посреди передачи бита.
- **0×08 Start** Был сделан старт. Теперь мы решаем что делать дальше, например послать адрес ведомого
- **0×10 ReStart** Был обнаружен повторный старт. Можно переключиться с записи на чтение или наоборот. От логики зависит.
- **0×18 SLA+W+ACK** Мы отправили адрес с битом записи, а в ответ получили ACK от ведомого. Значит можно продолжать.
- **0×20 SLA+W+NACK** Мы отправили адрес с битом записи, а нас послали NACK. Обидно, сгенерим ошибку или повторим еще раз.
- **0×28 Byte+ACK** Мы послали байт и получили подтверждение, что ведомый его принял. Продолжаем.
- **0×30 Byte+NACK** Мы послали байт, но подтверждение не получили. Видимо ведомый уже сыт по горло нашими подачками или он захлебнулся в данных. Либо его ВНЕЗАПНО посреди передачи данных украли инопланетяне.
- **0×38 Collision** А у нас тут клановые разборки — пришел другой мастер, по хамски нас перебил, да так, что мы от возмущения аж заткнулись. Ничего I'l be back! До встречи через n тактов!
- **0×40 SLA+R+ACK** Послали адрес с битом на чтение, а ведомый отозвался. Хорошо! Будем читать.
- **0×48 SLA+R+NACK** Крикнули в шину «Эй ты, с адресом XXX, почитай нам сказки» А в ответ «Иди NACK!» В смысле на запрос адреса с битом чтения никто не откликнулся. Видимо не хотят или заняты. Также может быть никого нет дома.
- **0×50 Receive Byte** Мы приняли байт. И думаем что бы ответить ведомому. ACK или NACK.

- **0×58 Receive Byte+NACK** Мы приняли байт от ведомого и сказали ему «иди NACK!» И он обиженый ушел, освободив шину.

Коды состояний для режима Slave:

- **0×68 Receive SLA+W LP** Мы были мастером, трепались с подчиненными по шине. И тут появляется на шине другой, более равный, мастер, перебивает нас и молвит «Уважаемый XX, а не возьмете ли вы вот эти байтики…» Чтож, он круче. Придется бросать передачу и брать его байты себе.
- **0×78 Receive SLA+W LP Broadcast** Были мы, как нам казалось, самыми крутыми мастерами на шине. Пока не появился другой мастер и перебив нас прогундосил на всю шину «Эй, слышь тыыы. Слушай сюда» Девайсы-лохи, с неотключенными широковещательными запросами подчиняются. Остальные отмораживаются и всякое Broadcast-быдло игнорируют.
- **0×60 Receive SLA+W** Сидим на шине, никого не трогаем, ни с кем не общаемся. А тут нас по имени… Конечно отзовемся :)
- **0×70 Receive SLA+W Broadcast** Ситуация повторяется, но на этот раз слышим уже знакомое нам «Слышь, тыыыы». Кто? К кому? Игнорируем Broadcast запросы? Или нет? Зависит от моральных качеств программы.
- **0×80 Receive Byte & 0×90 Receive Byte Broadcast** Принимаем байты. От кого и в каком виде не важно. Решаем что сказать Давай еще (ACK) или «Иди NACK». Тут уже по обстоятельствам.
- **0×88 Receive Last Byte & 0×98 Receive Last Byte Broadcast** Приняли последний байт и распихиваем по карманам.
- **0xA0 Receive ReStart** Ой у нас Повторный старт. Видимо то что пришло в первый раз был таки адрес страницы. А сейчас пойдут данные…
- **0xB0 Receive SLA+R LP** Слали мы что то слали, а тут нас перебивает другой мастер, обращается по имени и говорит «А ну XX зачитай нам что нибудь из Пушкина» Что делать, приходится читать.
- **0xA8 Receive SLA+R** Либо просто к нам какой то другой мастер по имени обращается и просить ему передать байтиков.
- **0xB8 Send Byte Receive ACK** Ну дали мы ему байт. Он нам ACK. А мы тем временем думаем слать ему еще один (последний) и говорить «иди NACK». Или же у нас дофига их и можно еще пообщаться.
- **0xC0 Send Last Byte Receive NACK** Дали мастеру последний имеющийся байт, а он нам «иди NACK». Хамло. Ну и хрен с ним. Уходим с шины.
- **0xC8 Send Last Byte Receive ACK** Дали мастеру последний имеющийся байт, а он требует еще. Но у нас нету, так что разворачиваемся и уходим с шины. А он пусть карманы воздухом наполняет (в этот момент мастер начнет получать якобы от slave 0xFF байты, на самом деле это просто чтение висящей шины).

Из каждого состояния можно выйти по фиксированному для каждого состояния пути. Как и куда выходить зависит от того, что нам нужно сделать. Какие данные послать.

Путь определяется битами регистра TWCR и управляется он только вручную!

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	—-	TWIE
7	6	5	4	3	2	1	0

- TWINT — флаг прерывания. Сброс его означает что конечный автомат провернется дальше, а прерывание будет снова уловлено.
- TWEA — Enable ACK. Разрешение ответа ACK. Если его включить, то автомат TWI будет отзываться на свой адрес, а также давать ACK во всех случаях когда этого требует протокол. Скажем, после прием байта. Если нам надо послать NACK то бит не ставим.
- TWSTA — Сформировать старт. Причем не факт, что прямо сейчас. От текущего состояния зависит. Например, если этот бит поставить в 0x38 состоянии, то старт будет сформирован когда шина освободится. Контроллер TWI умный и сам все хорошо знает.
- TWSTO — сделать Stop. Опять же аналогично. После записи в этот бит модуль отваливается от сессии передачи. МК становится неадресованным ведомым.
- TWWC — конфликт записи. Сигнализирует о том, что кто то из прикладной программы записал в TWDR данные. Тогда как в TWDR полагается писать только при поднятом TWINT (в нашем случае это будет толкьо в обработчике прерывания).
- TWEN — блок TWI включен. Только и всего.
- TWIE — прерывания от TWI разрешены.

Простой пример — оптправка данных в EEPROM. По правилам шины i2c она должна выглядеть так:

Master	Start	SLA+W		Byte		Byte		Stop
Slave			ACK		ACK		ACK	

Какова последовательность действий? Ну первым делом мы все подготовим, запишем в нужные места данные, чтобы контроллер знал потом откуда все хватать. Считаем что это уже сделано.

Start
В прикладной программе инициируем обмен по шине. Для этого всего то надо установить в TWCR биты:

- TWSTA — сказав, что мы формируем старт.
- TWINT — иначе ничего не завертится.
- TWEN — понятно зачем. Включить обязательно
- TWIE — прерывания тоже нам нужны.

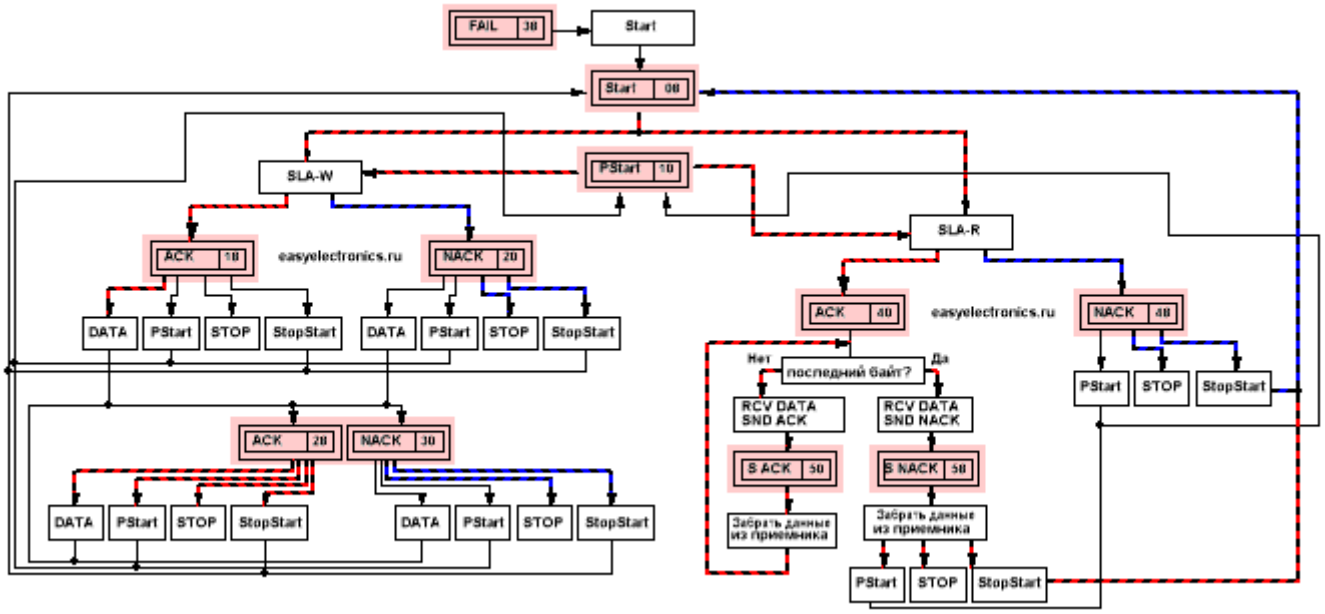
0×08
А дальше, как только блок TWI родит на шине Start, мы окажемся в прерывании TWI с кодом 0x08 и тут смотрим что надо сделать дальше. А дальше нам надо отправить SLA+W. SLA+W мы записываем в TWDR и даем приказ заслать это в шину, поставив биты
TWEN, TWIE, TWINT остальные не нужны. Т.к. это не старт, не стоп, и, для простоты, ведомым мы не являемся в принципе, а значит нам ACK никому слать не надо. Запуливаем эту комбинацию в TWCR и выходим из прерывания.

0×18
Спустя какое то время, когда блок TWI срыгнет адрес на шину, мы вновь окажемся в прерывании. Но уже с кодом 0x18 (или 0x20 если нам никто не ответил). Теперь надо решать что дальше делать. Надо послать байт данных. Суем данные в TWDR из условленного места. И снова проворачиваем механизм, выставив флаги. На этот раз нужен тоже только TWEN, TWIE и TWINT.
Блок TWI начнет отправлять данные по шине. Как закончит опять сгенерит прерывание, но уже с кодом 0x28

0x28
Мы послали байт и попали сюда. Код 0x28 (или код 0x20 если ведомый послал нас NACK). Теперь надо решать что делать дальше. У нас там по плану еще один байт послать. Суем его в TWDR и шлем (TWEN, TWIE и TWINT).

0x28
Опять в прерывании, опять с кодом 0x28. А байты то кончились. Больше ничего слать не надо. Значит пошлем СТОП и прекратим обмен. Не проблема (TWEN, TWIE, TWINT и, главное, TWSTO). Теперь где нибудь выставим флажок или пошлем событие о том, что передача успешно завершена, а затем, с чистой совестью, выйдем из прерывания в фоновую программу.

Сложно? Вот и я думаю, что не очень. Особенно когда все по полочкам разложишь.
А вот я рисовал когда то диаграмму движения по автомату с кодами ошибок и событий. Правда это не полный автомат, а только лишь режим Master и без арбитража



Покрупней [4]

Синии линии — косяки и сбои. Красные — спокойная работа когда все говоря ACK и не перечат.

А сам обработчик прерывания TWI пишется очень просто. Берешь и делаешь CASE в который забиваешь все возможные коды входа и варианты реакции на них. Получается чертовски много кода, но кусочки сами по себе простые. И за раз выполняется всего одна ветвь, а значит работает это ОЧЕНЬ быстро.

Причем не обязательно реализовывать все-все-все коды. Например, если у нас только Master то вся группа кодов для Slave не имеет смысла. Если мастер один, то вариантов развития с перехватом шины другим мастером тоже не будет. Или если проц работает только Slave то оставляем лишь Slave функции. Которые тоже можно почикать, убрав, например, прием байта если нам байты только отсылать. В общем, кромсай@отрезай. Главное затерминировать через Break или через Default остальные пути развития сценария. На всякий пожарный.

Я же, выполнил полностью весь автомат. С возможностями как Slave, так и Master. С возможностью перехвата обмена другим мастером. Не заморачивался только с широковещательными пакетами и повторным стартом для Slave режима.

Процесс
Итак, как происходит обмен. Так как у меня в проекте поднят диспетчер, то вся обработка начальных/конечных событий сделана через него.

Обмен для мастера состоит из пунктов:

- 1. Подготовка данных и формирование первичного старта. Т.е. все данные загружаются в нужные буферы, все индексы буферов выставляются на нули, записываются нужные адреса Slave и, если нужно, адреса страниц памяти, число передаваемых байт, число байт в адресе страницы. Требуемый режим обмена и тыды. В общем всю необходимую инфу для передачи данных. А также задаю адрес точки выхода из автомата. И запускаю конечный автомат.
- 2. Конечный автомат отрабатывает в соответстветствии с выбранным режимом передачи и на излете запускает выходную задачу.
- 3. Выходная задача (мы еще в прерывании TWI, поэтому действуем быстро-быстро, не тупим), считывая коды ошибок, проверяет успешность передачи, если что не так делает перезапуск посредством заброса функции первичного старта через службу таймеров на повторное исполнение. Скажем через 50мс. Если все хорошо, то ставим на конвейер диспетчера задачу следующую по алгоритму работы всей проги.

Обмен для слейва попроще:

- 1. Инициализируем Slave, заполняем мастер-адреса. Заполняем буфера выходными данными (если такие могут быть). Включаем ACK, включаем прерывания TWI, включаем сам TWI. Задаем точку выхода из конечного автомата. Ждем когда к нам соизвоят постучаться.

- 2. Если обращение произошло, то автомат его обрабатывает, заливает/сливает данные с выходных буферов. А по окончании обмена выполняет выходную задачу.
- 3. Выходная задача (мы еще в прерывании TWI, поэтому действуем быстро-быстро, не тупим) обрабатывает сгребает данные из входящих буферов, сигнализирует фоновой программе или запускает через диспетчер задачу исходя из пришедших данных.

Причем во время работы мастера может произойти вызов его как слейва и тогда возникнет переход от одного обмена к другому. Туда и обратно. С повторной попыткой обмена.

Крутилки, ручки и кнопочки
Опишу вначале все барахло что у меня там есть. Тут кратко. Подробней в исходниках, в комментариях.
Опции

- **i2c_PORT & i2c_DDR** — Порт где сидит нога TWI
- **i2c_SCL & i2c_SDA** — Биты соответствующих выводов
- **i2c_MasterAddress** — Адрес на который будем отзываться в роли Slave
- **i2c_i_am_slave** — Если мы еще и слейвом работаем то 1. Если только мастером то 0.
- **i2c_MasterBytesRX** — Величина принимающего буфера режима Slave, т.е. сколько байт жрем.
- **i2c_MasterBytesTX** — Величина Передающего буфера режима Slave , т.е. сколько байт отдаем за сессию.
- **i2c_MaxBuffer** — Величина буфера Master режима RX-TX. Зависит от того какой длины строки мы будем гонять
- **i2c_MaxPageAddrLgth** — Максимальная величина адреса страницы. Обычно адрес страницы это один или два байта. Зависит от типа ЕЕПРОМ или другой микросхемы.

Стартовые данные

- **i2c_Do** — основная переменная состояния. Она флаговая. В нее вписываем флаги режима работы, из нее берем коды ошибок
- **i2c_InBuff** — буфер приема в режиме Slave. Т.е. если мы работаем как ведомыми и нас вызвал мастер, то он впишет свои байтики сюда. Причем впишет не больше чем указано. Т.к. на последнем байте мы его пошлем NACK и откажемся принимать мастерские подачки.
- **i2c_OutBuff** — а это выходной буфер ведомого. Когда мастер нас позовет и скажет «Есть чо?», то мы ему выгрузим данные отсюда. Таким образом, общение в качестве ведомого ведется через эти буфера. Например, в мультироцессорной робо системе, где есть голова, а есть ведомое шасси, то контроллер шасси может хранить в InBuff значения требуемой скорости и текущую команду. А в OutBuff показания датчиков и, например, показания с АЦП висящем на блоке аккумуляторов. И при обмене данными с головным контроллером он сливает ему OutBuff, а от него принимает InBuff
- **i2c_Buffer**— Это выходной буфер мастера. Если наш контроллер в качестве мастера ломанется куда-нибудь, то он сюда загрузит всю необходимую инфу и пнет ее слейву.
- **i2c_ByteCount** — важный параметр. Число передаваемых байт мастером. Можно сделать константой (проще, но не гибче) я сделал переменной, чтобы гонять данные туды сюды.
- **i2c_SlaveAddress** — Адрес подчиненного. То куда мы будем обращаться в режиме мастера.
- **i2c_PageAddress** — Буфер адреса страниц. Если вы помните режим обмена со всякими ЕЕПРОМ, то для чтения произвольного байта мы должны сначала записать в ЕЕПРОМ адрес этого байта, потом, вызвав ReStart считать данные. Вот тут у нас буфер где хранится адрес страницы для чтения ЕЕПРОМ. Сделан он буфером для универсальности. Т.к. у некоторых ЕЕПРОМ адрес страницы двухбайтный, а у некоторых однобайтный. Более того, есть куча устройств которые выглядят с точки зрения мастера как EEPROM, но не являются ими. Например часы RTC или цифровые акселерометры. У них, обычно, адрес страницы однобайтный.
- **i2c_PageAddrCount** — Число байт в адресе страницы для текущего Slave

Указатели выхода из автомата. У меня выход из автомата сделан на функцию, адрес которой передается при вызове Master обмена или при включении Slave ожидания. Их три вида.

- **IIC_F MasterOutFunc**— выход из автомата для мастера
- **IIC_F ErrorOutFunc**— выход из автомата в результате ошибки в режиме Master
- **IIC_F SlaveOutFunc**— выход из автомата в режиме Slave

Обычно я первые два обьединяю под одну функцию, а там уже проверяю на ошибки. Но, для оптимальности, можно и разделить их. Причем сам выход записан в исходнике в форме макроса. Что позволяет, не лазая в дебрях автомата подправить в хидере этот макрос на что угодно. Например, на выставление флажка, если у вас не диспетчер, а флаговый автомат.

Режимы

- **i2c_sarp** (Start-Addr_R-Read-Stop) Это режим простого чтения. Например из слейва или из епрома с текущего адреса
- **i2c_sawp** (Start-Addr_W-Write-Stop) Это режим простой записи. В том числе и запись с адресом страницы.
- **i2c_sawsarp** (Start-Addr_W-WrPageAdr-rStart-Addr_R-Read-Stop) Это режим с предварительной записью текущего адреса страницы. Например, через нее из EEPROM читают данные.

Коды ошибок

- **i2c_Err_NO** All Right! — Все окей, передача успешна.
- **i2c_ERR_NA** — Device No Answer Слейв не отвечает. Т.к. либо занят, либо его нет на линии.
- **i2c_ERR_LP** — Low Priority нас перехватили собственным адресом, либо мы проиграли арбитраж
- **i2c_ERR_NK** — Received NACK. End Transmittion. Был получен NACK. Бывает и так.
- **i2c_ERR_BF** — BUS FAIL Автобус сломался. И этим все сказано. Можно попробовать сделать переинициализацию шины

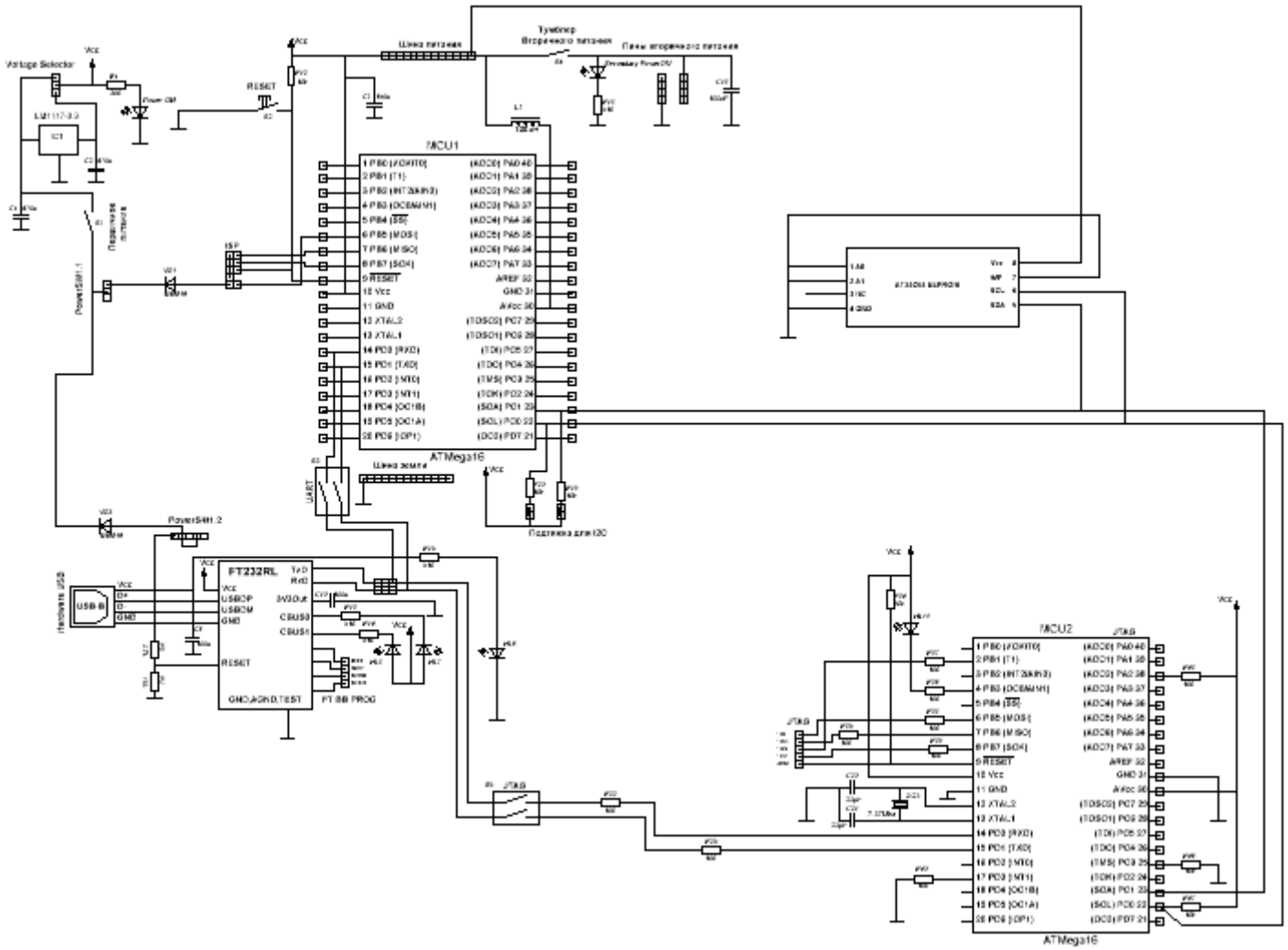
Также есть флаги состояний

- **i2c_Interrupted** — передачу мастера прервали запросом собственного адреса от другого мастера
- **i2c_Busy** — Передатчик занят другим процессом отправки/приема, руками не трогать.

…
— А есть любовь к доминированию или, наоборот, подчинению. Это называется садомазохизм.
— Слайды давай, слайды!
…

Слайды
А сейчас, дабы не быть голословным, я покажу вам извращенную групповую садомазо оргию с участием одного мазохиста и двух садо-мазохистов.

Итак, у нас есть одна большая крова… эмм шина IIC. На ней висят два контроллера и одна EEPROM.



[Посмотреть крупней](#) [5]

Контроллер1 получет из космоса бредовую мысль (байт по UART) и жестко домогась(master) до EEPROM(slave) забивает эту мысль ей в башку по адресу 0x00FF. Но его не отпускает. Он решает поделиться мыслью и с Контроллером2. Но не просто так, а с подвывертом, будто это не ему в голову пришла бредовая мысль. Контроллер1(Master) ломится к Контроллеру2(Slave) и говорит ему — чувак, там у EEPROM по адресу 0x00FF в бошке засела умная мысль. Нуко достань! Контроллер2, получив адрес (0x00FF) жесто домогается (Master) до EEPROM(slave) вбивает ей в мозги адрес, потом вытягивает бредовую мысль (тот самый байт). Додумывает к ней подробностей (прибавляет 1) и ломится(Master) со своей находкой к Контроллеру1(slave). Контроллер1, внемлет второму, принимает его бредовые идеи, и отправляет их обратно в космос (по UART). Все успокаиваются до новой посылки из космоса.

Причем все, кроме забитой еепромки, страдают манией и если кто то не отвечает — начинают настойчиво до него долбиться до тех пор, пока получат желаемое. Епромка же отличается тормознутостью. Умная мысль вгоняет ее в такой ступор (а шо ви хотели, flash медленная штука), что она целых 20мс не отвечает на домогания контроллера2. А он же ломится до тех пор, пока не ответят.

Код
Инициализация Master режима заключается лишь в настройке скорости передачи, да включении.

```
1 void Init_i2c(void)                    // Настройка режима мастера
2 {
3 i2c_PORT |= 1<<i2c_SCL|1<<i2c_SDA;     // Включим подтяжку на ноги, вдруг юзер на
4 резисторы пожмотился
5 i2c_DDR &=~(1<<i2c_SCL|1<<i2c_SDA);
6
7 TWBR = 0x80;                           // Настроим битрейт
8 TWSR = 0x00;
```

```
}
```

Для Slave режима надо задать параметры ведомого. Адреса, а также точки выхода. Ну и включить передатчик соответствующим образом. Чтобы принимал свой адрес и отдавал ACK. Разумеется должны быть разрешены прерывания

```
void Init_Slave_i2c(IIC_F Addr)        // Настройка режима слейва (если нужно)
{
TWAR = i2c_MasterAddress;      // Внесем в регистр свой адрес, на который будем
отзываться.
                               // 1 в нулевом бите означает, что мы отзываемся на
широковещательные пакеты
SlaveOutFunc = Addr;           // Присвоим указателю выхода по слейву функцию выхода

TWCR =  0<<TWSTA|
        0<<TWSTO|
        0<<TWINT|
        1<<TWEA|
        1<<TWEN|
        1<<TWIE;               // Включаем агрегат и начинаем слушать шину.
}
```

Код библиотеки лучше смотреть в исходниках примеров, т.к. он слишком громоздок. Хотя и прост. Т.к. там нет ни одного замороченного алгоритма, а самое сложное что там есть — проверка битов режима. Т.к. все остальное делает автомат TWI.
IIC_ultimate.c [6]
IIC_ultimate.h [7]

Опишу лучше работу с ней на примере управляющей логики нашей садомазо группы

Контроллер 1

Он начинает первый.

Его main функция выглядит вообще просто:

```
int main(void)
{
InitAll();                             // Инициализируем периферию
Init_i2c();                            // Запускаем и конфигурируем i2c
Init_Slave_i2c(&SlaveControl);         // Настраиваем событие выхода при сработке как
Slave

/*
WorkIndex=0;                   // Лог с начала
WorkLog[WorkIndex]=1;          // Записываем метку старта
WorkIndex++;
*/

InitRTOS();                    // Инициализируем ядро
RunRTOS();                     // Старт ядра.

_delay_ms(1);                  // Небольшая выдержка, чтобы второй контроллер успел
встать на адресацию


while(1)                       // Главный цикл диспетчера
{
wdt_reset();                   // Сброс собачьего таймера
TaskManager();                 // Вызов диспетчера
}
```

```
26
   return 0;
   }
```

Как видим — нет ничего кроме инициализации. Никакие задачи не страртуют. Экшн начинается когда приходит байт по UART

```
1 ISR(USART_RXC_vect)
2 {
3 UART_RX = UDR;                  // Сгребаем принятый байт в буфер
4 SetTask(StartWrite2EPP);         // Запускаем процесс записи в ЕЕПРОМ.
5 }
```

Сохраняется принятый байт и через диспетчер запускается процедура отправки байта в EEPROM

```
1 void StartWrite2EPP(void)
2 {
3 if (!i2c_eep_WriteByte(0xA0,0x00FF,UART_RX,&Writed2EEP))      // Если байт не записался
4         {
5         SetTimerTask(StartWrite2EPP,50);                     // Повторить попытку через
6 50мс
7         }
  }
```

Функция **i2c_eep_WriteByte** отдает 1 если шина была свободна и процесс пошел. Если шина была занята обменом с другими участниками групповухи, то повтор попытки. Просто перезагрузка задачи через таймер. В параметрах функции у нас адрес EEPROM на шине i2c (0xA0) и адрес страницы по которой мы будем писать данные (0x00FF), передаваемый байт из UART_RX, а также адрес процедуры выхода из автомата (Writed2EEP).

Когда автомат отработает все, мы туда и попадем

```
1  void Writed2EEP(void)
2  {
3  i2c_Do &= i2c_Free;                    // Освобождаем шину
4
5  if(i2c_Do & (i2c_ERR_NA|i2c_ERR_BF))   // Если запись не удалась
6          {
7          SetTimerTask(StartWrite2EPP,20);       // повторяем попытку
8          }
9  else
10         {
11         SetTask(SendAddrToSlave);      // Если все ок, то идем на следующий
12         }                              // Пункт задания - передача данных слейву 2
13 }
```

Тут все просто. Проверим, что все ок и переходим к другому Slave девайсу.

```
1  // Обращение к SLAVE контроллеру
2  void SendAddrToSlave(void)
3  {
4  if (i2c_Do & i2c_Busy)                         // Если передатчик занят
5          {
6          SetTimerTask(SendAddrToSlave,100);     // То повторить через 100мс
7          return;
8          }
9
10 i2c_index = 0;                          // Сброс индекса
11 i2c_ByteCount = 2;                      // Шлем два байта
12
13 i2c_SlaveAddress = 0xB0;                // Адрес контроллера 0xB0
```

```
14
15 i2c_Buffer[0] = 0x00;                  // Те самые два байта, что мы шлем подчиненному
16 i2c_Buffer[1] = 0xFF;
17
18 i2c_Do = i2c_sawp;                     // Режим = простая запись, адрес+два байта данных
19
20 MasterOutFunc = &SendedAddrToSlave;    // Точка выхода из автомата если все хорошо
21 ErrorOutFunc = &SendedAddrToSlave;     // И если все плохо.
22
23 TWCR =  1<<TWSTA|
24         0<<TWSTO|
25         1<<TWINT|
26         0<<TWEA|
27         1<<TWEN|
28         1<<TWIE;                      // Поехали!
29
30 i2c_Do |= i2c_Busy;                    // Шина занята!
31 }
```

Как я и описывал в разделе опций, мы тут полностью определяем параметры будущей передачи. Сколько байт передаем (i2c_ByteCount) кому передаем (i2c_SlaveAddress), что передаем (i2c_Buffer), а также задаем режим передачи — простая запись (i2c_sawp) . И точку выхода (SendedAddrToSlave).

Когда автомат отработает, то мы вываливаемся в принимающую задачу.

```
1 void SendedAddrToSlave(void)
2 {
3 i2c_Do &= i2c_Free;                              // Освобождаем шину
4
5 if(i2c_Do & (i2c_ERR_NA|i2c_ERR_BF))             // Если адресат нас не услышал или был сбой
6 на линии
7         {
8         SetTimerTask(SendAddrToSlave,20);       // Повторить попытку
9         }
  }
```

Тут все просто. Главное проверить, что все прошло хорошо. И перезапустить в случае косяка.

Остается подчиненная часть. Тут все еще проще. Только одна функция, точнее точка выхода из автомата:

```
1 void SlaveControl(void)
2 {
3 i2c_Do &= i2c_Free;                    // Освобождаем шину
4 UDR = i2c_InBuff[0];                   // Выгружаем принятый байт
5 }
```

Освобождаем шину и загребаем байт из приемного буфера.

Контроллер 2

Тут тоже все подобным образом, но в другом порядке. Тон тут задает не прерывание UART, а приход адресных байтов от Контроллера 1.

```
1  int main(void)
2  {
3  InitAll();                                  // Инициализируем периферию
4  Init_i2c();                               // Настроили мастер режим
5  Init_Slave_i2c(&SlaveControl);             // Настроили слейв режим
6
7  /*
```

```
8  WorkLog[WorkIndex]=1;
9  WorkIndex++;
10 */
11
12 InitRTOS();                              // Инициализируем ядро
13 RunRTOS();                               // Старт ядра.
14
15
16
17 while(1)                                 // Главный цикл диспетчера
18 {
19 wdt_reset();                             // Сброс собачьего таймера
20 TaskManager();                           // Вызов диспетчера
21 }
22
23 return 0;
24 }
```

Только настроили точку выхода из TWI (SlaveControl)

```
   void SlaveControl(void)                    // Точка выхода из автомата слейва
1  {
2  i2c_Do &= i2c_Free;                       // Осовобождаем шину
3  SetTask(ReadEEPROM);                     // Готовим запись в ЕЕПРОМ
4  }
5
6  Байты получены. Можно запускать чтение из EEPROM.
7
8  void ReadEEPROM(void)                                    // Читаем из еепром
9  {
10 u16 Addr;
11
12 Addr = (i2c_InBuff[0]<<8)|(i2c_InBuff[1]);                    // Адрес возьмем из буфера
13 слейва
14
15 if (!i2c_eep_ReadByte(0xA0,Addr,1,&EepromReaded) )       // Читаем
16    {
17    SetTimerTask(ReadEEPROM,50);                                // Если процесс не пошел
18 (шина занята), то повтор через 50мс.
19    }
   }
```

Все аналогичным же образом. Разница лишь в том, что тут мы читаем из памяти. И точка выхода у нас другая (EepromReaded)

```
   void EepromReaded(void)                   // Была попытка чтения
1  {
2  i2c_Do &= i2c_Free;                       // Освобождаем шину
3
4  if(i2c_Do & (i2c_ERR_NA|i2c_ERR_BF))                 // Ошибки при четнии были?
5     {
6     SetTimerTask(ReadEEPROM,20);            // Тогда повтор
7     }
8  else
9     {
10    ReadedByte = i2c_Buffer[0];                       // Иначе считаный байт из буфера
11 копируем в переменную
12    SetTask(SendByteToSlave);                // И запускаем отсылку ее контроллеру 1
13    }
14 }
```

Когда EEPROM будет успешно прочитана, то мы загоним задачу отправки байта контроллеру 1 (SendByteToSlave) . Вот эта задача:

```
   void SendByteToSlave(void)                    // Возвращаем контроллеру 1 его байт
1  {
2  if (i2c_Do & i2c_Busy)                        // Если шина занята
3        {
4        SetTimerTask(SendByteToSlave,100);        // То повторяем попытку
5        return;
6        }
7
8  i2c_index = 0;                                // Сброс индекса
9  i2c_ByteCount = 1;                            // Шлем 1 байт
10
11 i2c_SlaveAddress = 0x32;                      // Адрес контроллера 1 на шине
12
13 i2c_Buffer[0] = ReadedByte+1;                 // Загружаем в буфер число, увеличив его на
14 1.
15                                               // +1 чтобы понять, что число прошло через
16 МК и было обработано
17
18
19 i2c_Do = i2c_sawp;                            // Режим - простая запись
20
21 MasterOutFunc = &SendedByteToSlave;           // Задаем точку выхода
22 ErrorOutFunc = &SendedByteToSlave;
23
24 TWCR =  1<<TWSTA|
25         0<<TWSTO|
26         1<<TWINT|
27         0<<TWEA|
28         1<<TWEN|
29         1<<TWIE;                              // Поехали!
30
31 i2c_Do |= i2c_Busy;                           // Флаг занятости поставим
   }
```

Все полностью по аналогии.

```
1  void SendedByteToSlave(void)                  // Байт был послан
2  {
3  i2c_Do &= i2c_Free;                           // Осовбождаем шину
4
5
6  if(i2c_Do & (i2c_ERR_NA|i2c_ERR_BF))          // Если посылка не удалась
7     {
8     SetTimerTask(SendByteToSlave,20);          // Пробуем еще раз.
9     }
10 }
```

И все. Тупо ждем следующего экшна. Т.е .в принцип работы заложена основа, что у нас обмен делается посредством связки из двух функций. Запускающей и выходящей.

EEPROM

Поскольку ей в мозги не заглянешь, то покажу тут содержимое библиотечки, через которую мы к ней стучались. Тоже накорябал на днях. Если IIC_ultimate.c это своего рода полуфабрикат, то это уже ближе к финальному продукту, для непосредственной работы.

Там все очень и очень просто, смотрите сами:

```
#include <i2c_AT24C_EEP.h>


#define HI(X) (X>>8)                    // Макросы разделения слова на байты
#define LO(X) (X & 0xFF)

u08 i2c_eep_WriteByte(u08 SAddr,u16 Addr, u08 Byte, IIC_F WhatDo)
{

if (i2c_Do & i2c_Busy) return 0;        // Если шина занята, то возвращаем 0

i2c_index = 0;                          // Зануляем индексы
i2c_ByteCount = 3;                      // Запись байта всего делается за три байта.
                                        // Два байта -- адрес страницы и байт данных.

i2c_SlaveAddress = SAddr;               // Какой именно EEPROM будем слать.


i2c_Buffer[0] = HI(Addr);               // В буфер кладем адрес страницы
i2c_Buffer[1] = LO(Addr);               // Старший и младший байты, по очереди. Старший
впереди
i2c_Buffer[2] = Byte;                   // Ну и байт данных

i2c_Do = i2c_sawp;                      // Режим простой записи

MasterOutFunc = WhatDo;                         // Точка выхода которую нам передадут
ErrorOutFunc = WhatDo;

TWCR = 1<<TWSTA|0<<TWSTO|1<<TWINT|0<<TWEA|1<<TWEN|1<<TWIE;    // ПЦ!

i2c_Do |= i2c_Busy;                     // Занимаем шину

return 1;                               // Возвращаем 1, мол процесс пошел.
}

// Это чтение из EEPROM Тут чуточку сложней
u08 i2c_eep_ReadByte(u08 SAddr, u16 Addr, u08 ByteNumber, IIC_F WhatDo)
{
if (i2c_Do & i2c_Busy) return 0;        // Если шина занята

i2c_index = 0;                          // Индексы в ноль
i2c_ByteCount = ByteNumber;             // Число считываемых байт

i2c_SlaveAddress = SAddr;               // Адрес EEPROM на шине IIC

i2c_PageAddress[0] = HI(Addr);          // На этот раз грузим адрес страницы
i2c_PageAddress[1] = LO(Addr);          // В специальный буфер

i2c_PageAddrIndex = 0;                  // Обнуляем индексы
i2c_PageAddrCount = 2;                  // Адрес страницы из 2 байт

i2c_Do = i2c_sawsarp;                   // Режим чтения с заносом адреса

MasterOutFunc = WhatDo;                         // Точка выхода
ErrorOutFunc = WhatDo;

TWCR = 1<<TWSTA|0<<TWSTO|1<<TWINT|0<<TWEA|1<<TWEN|1<<TWIE;    //ПЦ!

i2c_Do |= i2c_Busy;                     // Заняли шину, чтобы никто не мешал.

return 1;                               // Вернули 1, мол процесс пошел.
}
```

Вот так вот.

Железо

Чтобы такое по быстрому слабать на коленке я заюзал свою маленькую прелесть… Дада вы уже догадались. Pinboard [8]. Там есть почти все необходимое для нашей вакханалии. Два контроллера и макетное поле в которое мы засадим микросхему памяти.

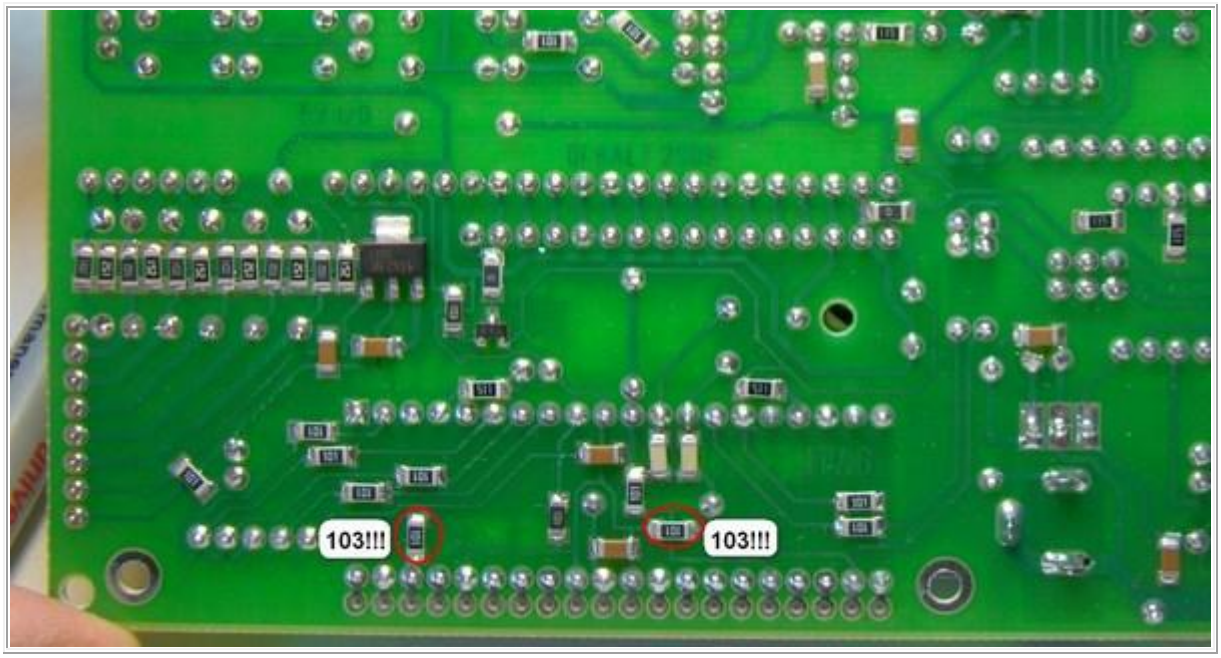
Один контроллер, правда, это JTAG адаптер. Но это не страшно. На борту есть FTBB программатор, поэтому нам ничего не стоит прошить его тем же бутлоадером, превратив в рядовой контроллер, подобный главному (обратно вернуть просто, достаточно запустить батник зашивки кода JTAG см. документацию по самопрограммированию [9])

Подключите FTBB выводы к порту прошивки JTAG контроллера и выполните батник **PBSelf16.cmd**после чего контроллер также начнет отвечать на бутлоадер. Надо будет только подсоединить его к порту джамперами JTAG (отключив переключатели UART). Да, для прошивки обеих контроллеров вы тумблерочками этими туда-сюда нащелкаетесь :)

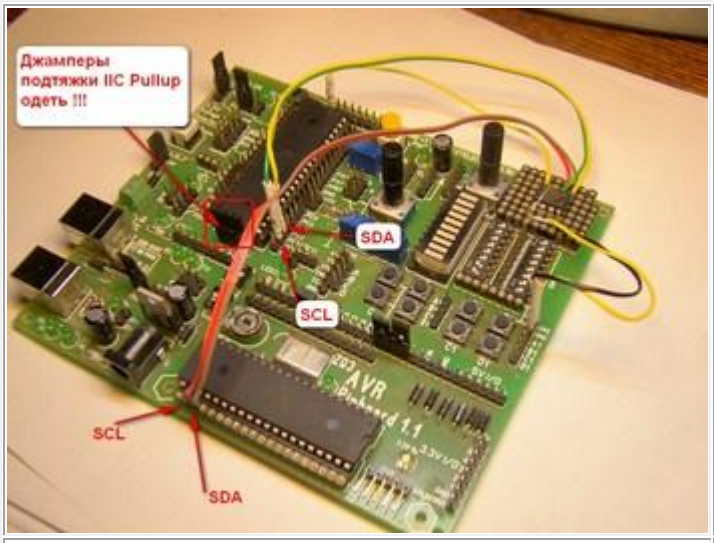
Потом снимите шлейфик и пользуйтесь им как главным, т.е. через бутлоадер. А чтобы вернуть JTAG адаптер на место то просто оденьте шлейф обратно, и запустите батник **PBSelfJT.cmd** и прошивка JTAG'а будет возвращена на место.

Также, в некоторых версиях возможно придется перепаять один резистор на плате. Дело в том, что изначально JTAG контроллер планировался как отладочный JTAG интерфейс и не более того. Однако куда интересней когда мы его можем использовать в любых целях. Но осознание этого пришло несколько позже :(. Так что в старых ревизиях версии 1.1 на линии SCL у нижней меги стоит резистор в 100ом, что слишком мало для подтяжки. Нога может не придавить и шина работать не будет. Его надо аккуратно выпаять и впаять туда резистор на 10к. Отпаивается smd резистор довольно просто — кладем на него паяльник плашмя, так чтобы прогреть его весь, да сдвигаем зубочисткой. В новых версиях там уже везде идут другие номиналы.

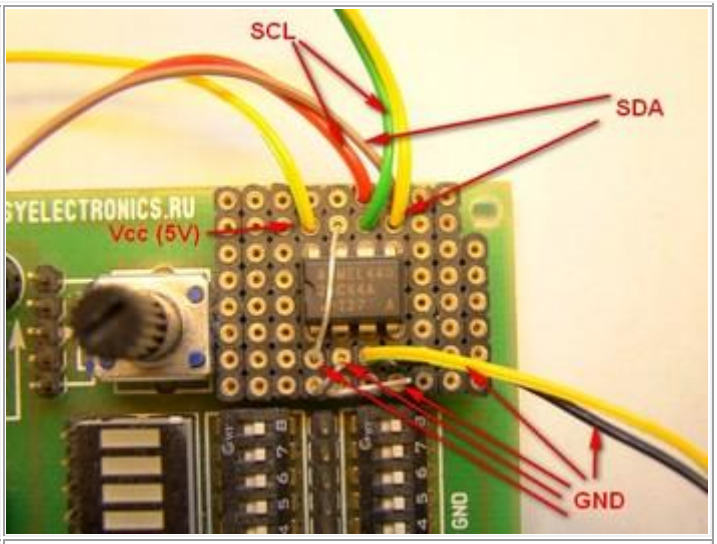
Резисторы которые надо заменить показаны на картинке:



Сборка схемы
Что к чему идет показано на фотках (кликабельны). Думаю все понятно. Благо схема то там — три с половиной проводка.



Увеличить [10]



Увеличить [11]

Ну и процесс работы

- **Архив с проектами под оба контроллера** [12]

Пара слов о отладке
В процессе написания было наделана куча багов, порой идиотских, но от этого не менее страшных. В процессе отладки активно применялся отладочный буфер. Его вы найдете закомментированым в коде. Это барахло вида:

```
1  if (WorkIndex <99)                                    // Если лог не переполнен
2  {
3          if (TWSR)                                     // Статус нулевой?
4                  {
5                  WorkLog[WorkIndex]= TWSR;             // Пишем статус в лог
6                  WorkIndex++;
7                  }
8          else
9                  {
10                 WorkLog[WorkIndex]= 0xFF;             // Если статус нулевой то вписываем
   FF
11
12                 WorkIndex++;
13                 }
   }
```

И прочие почеркушки в массив WorkLog.

Т.е. мы пишем все статусы TWSR с лог, а потом, спустя какое то время (я выбирал около 1с) сбрасываем его в терминал.

```
1  void LogOut(void)                                  // Выброс логов
2  {
3  u08 i;
4
5  WorkLog[WorkIndex]= 0xFF;
6  WorkIndex++;
7
8  for(i=0;i!=WorkIndex+1;i++)
9          {
10         UDR = WorkLog[i];
11         _delay_ms(30);
12         }
13 }
```

Напрямую в UART данные пихать нельзя. Т.к. TWI автомат щелкает гораздо быстрей чем UART может прожевать. И получится что что то потеряется, что то перепутается. А так мы получаем четкую историю работы программы. Понятно что происходит, где зацикливается передача, где не идет обмен.

Вот, например, мне Slave выдавал такие логи в хексах:

```
60 80 88 08 20 08 18 28 28 10 40 58 08 18 30
```

Сразу видно, по статусам, как прошел обмен. Как он получил байт от Master (60 80 88). Как попытался достучаться до EEPROM, но та была занята прожевыванием предыдущего байта (08 20). Как повторил попытку и успешно записал в нее адрес и считал байт (08 18 28 28 10 40 58), а потом отправил его другому контроллеру (08 18 30).

А отслеживать прохождение отдельных байт помогал осциллограф. Показывая что творится на шине, есть ли там NACK/ACK и когда идет ответы от Slave.

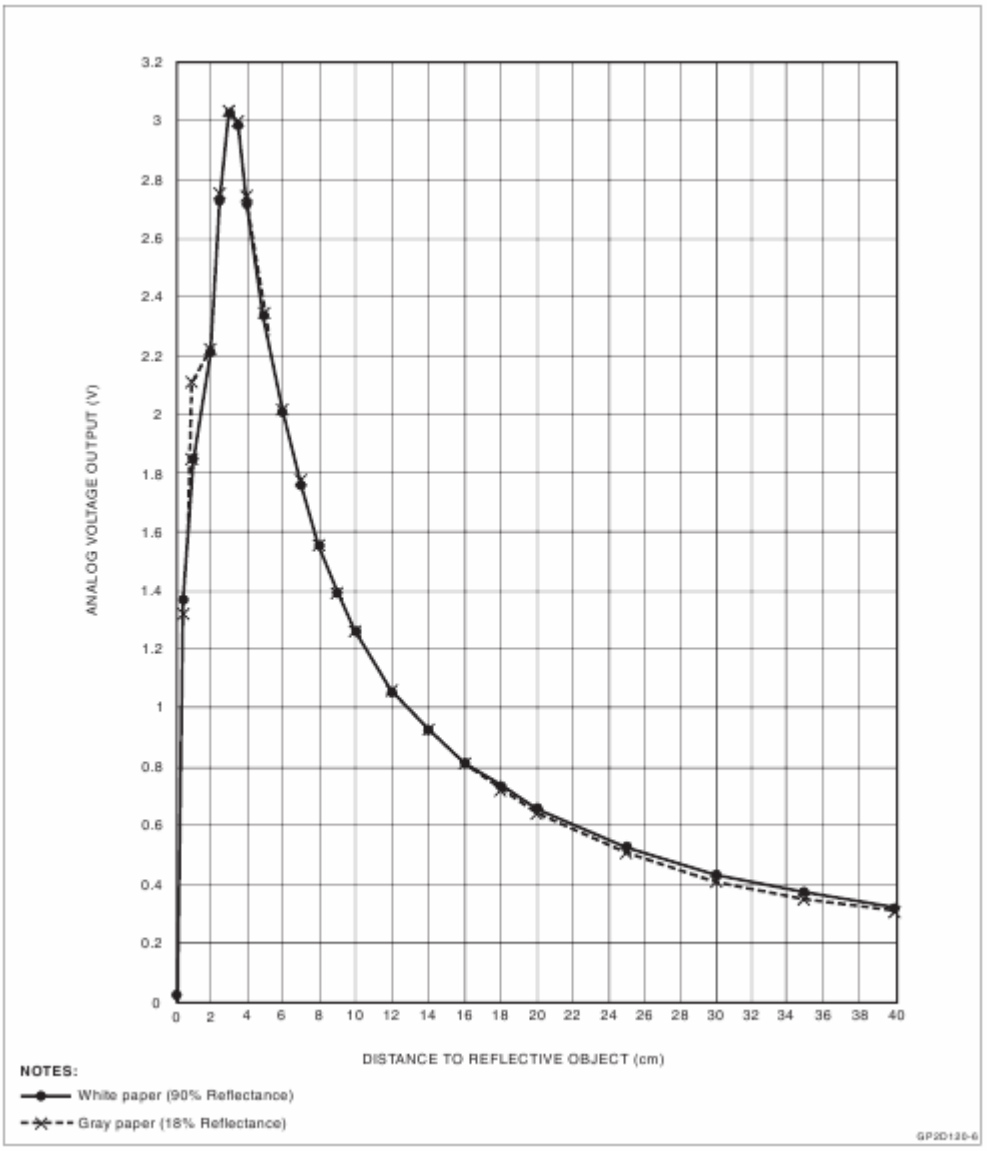
Кстати, пробовал, ради прикола, погонять пример в Proteus — выдал какую то неадекватную муть. I2C анализатор вообще половину обмена прошляпил. Хотя, по логам в терминалах, вторая половина все же был. Хотя и через задницу и не до конца. Плюс еще были интересные баги от того, что Proteus контроллер при старте запускает в нули , а реальный процессор может иметь неопределенное значение в некоторых регистрах. Так что был прикольный баг, когда прошивка в Proteus работает, а в реале нифига подобного. В общем, диодиком помигать сгодится, но пытаться отлаживать в нем что либо серьезное это надо быть сильно просветленным. Т.к. в железе все оно будет вести себя совсем по другому.

Доработки
Глюки я вроде бы поотлавливал. В разных режимах мастеров посталкивал. Но кое что надо доработать. Например, избавиться от полчища буферов и первести все на указатели. Чтобы в функции запуска/инициализации передвались только указатели на произвольный массив. Также было предложение все параметры согнать в структуры. Так что версию я может еще и обновлю. Но пока не до нее. Еще интересные вещи есть с которыми стоит поковыряться.

З.Ы.
Большое спасибо камраду Dcoder’у за ценные советы и идеи в процессе отладки.

AVR. Учебный Курс. Кусочно-линейная аппроксимация

Часто бывает так, что приходится обрабатывать жутко нелинейные величины, задаваемые каким-нибудь извращенным законом. Простейший пример — датчики расстояния SHARP GP2D12. Только поглядите на его характеристику:



Сам черт ногу сломит, а ведь нам бы неплохо иметь выход в человеческих величинах, ну или, хотя бы, линейно зависящие от расстояния. Что делать?

Вариантов тут, на самом деле, всего два. Первый очень быстрый, но жадный до памяти ПЗУ — табличный.
То есть мы просто берем и эту кривулину расписываем в памяти. Например, у нас с 8ми разрядного АЦП идет значение напряжения от 0 до 256, а мы на каждое значение создаем в памяти значение расстояния. Тогда с АЦП сразу гоним в индекс массива, где эти значения хранятся и получаем расстояние:

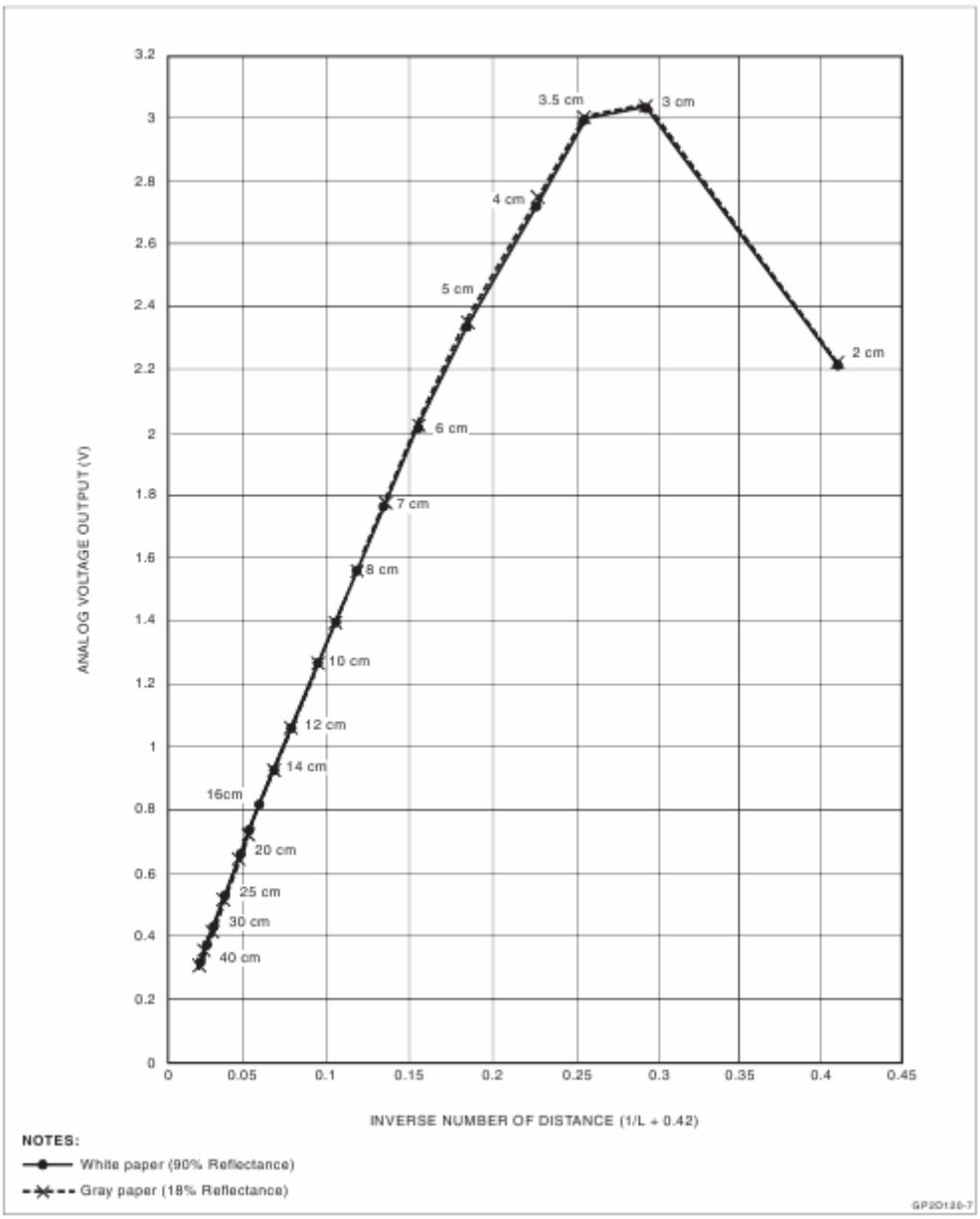
L=Curve[ADCH];

Недостаток один — прожорливость до памяти, растущая в геометрической прогрессии с ростом разрядности АЦП.

Вариант второй — написать функцию, переводящую одну величину в другую.
И вот тут можно поизвращаться. Дело в том, что если попытаться первести данные напрямую, как нам это советует даташит, на этот дальномер, то мы получим выражение вида:

L = k*X +b — уравнение прямой.

По даташиту видно, что характеристика становится линейной при выворачивании её наизнанку:



То есть у нас получается:

V = 1/L*k+b

- V — мы знаем, это показание с АЦП, помноженные на вес разряда: ADC * digit_weight. АЦП в данный момент имеет опорное напряжение в 3.3 вольта и 8 бит. Так что вес будет 3.3/255 = 0,013 вольта
- k — судя по графику, равно около 13.
- b — на вскидку, если продолжить график до нуля, выглядит примерно на 0.1

Получили:

0.013*ADC = 1/L*13 + 0.1

Путем применения технологий жуткого матана :) мы получаем чистую формулу расстояния:

L = 13/(0.013ADC-0.1)

Прикинув по исходному графику, убеждаемся что нигде ничего не забыли и не напутали.
Теперь покоцаем все на 0.013 и округлим результаты. Пока мы считаем сами нам доступна эта роскошь. МК же в целочисленных вычислениях округлять не умеет.

L = 1000/(ADC-8)

Вот, совсем красиво получилось. Одна непрятность — деление то у нас целочисленное, а значит будут потери округления. Т.е. при целочисленном вычислении 9.9 и 9.1 округляются до 9. Непорядок.

Наиболее точный результат целочисленного деления выражения A=B/C дает формула A = (B+C/2)/C

L = (1000+(ADC-8)/2)/(ADC-8)

В принципе, так можно и оставить и в таком виде скормить компилятору. Но можно и пооптимизировать. Во первых, что в первую очередь просится под нож, так это операция деления — самая мерзкая и тормозная из арифметических операций.

Как минимум одно деление можно оптимизировать:

```
1  u08 conversion (u08 _ADC)
2  {
3  u08 Half_DIV;
4  u16 Result;
5
6  if(_ADC<15) return 255;                // Очень далеко. Или ошибка датчика.
7
8  Half_DIV = _ADC-8;
9  Half_DIV >>=1;                  // Делим пополам сдвигом.
10
11 Result = 1000+(u16)Half_DIV;
12
13 return (u08)(Result/(_ADC-8));
14 }
```

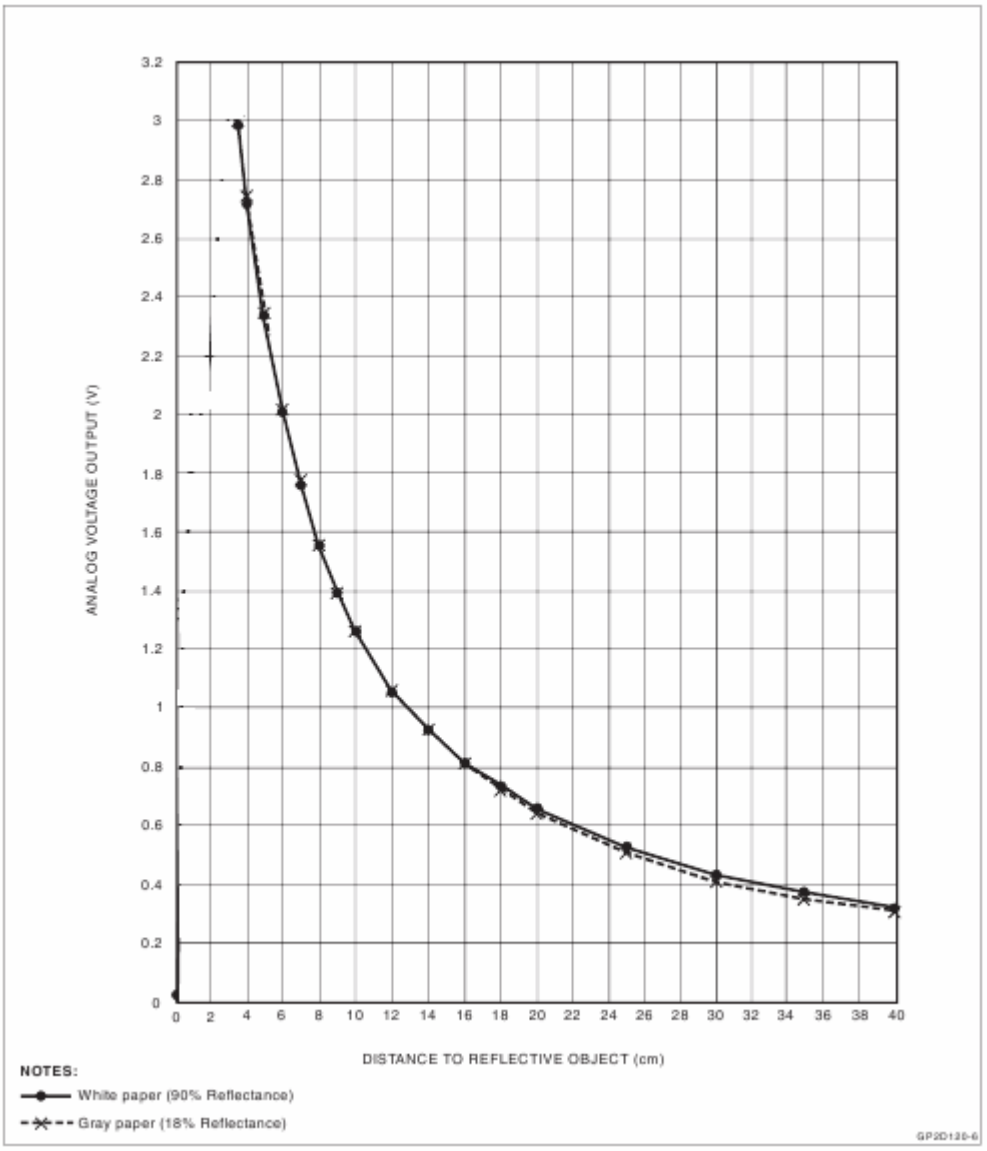
Зачем в самом начале проверку на минимальное значение? Да на всякий случай, чтобы не было деления на ноль, например. Или каких то космических значений. Видишь же, что характеристика датчика по напряжению не опускается ниже 0.2 вольт. А 0.2 вольта в наших показаниях АЦП это 15. Можно и вообще ограничить зону точкой четкого определения расстояния, скажем, 30 сантиметров. Это 0.4 вольта, тогда ограничивать можно сразу от 30 АЦПшных попугаев.

Аппроксимация кусочно-линейными функциями
Но это нам повезло с тем, что форма и параметры кривой оказались такими, что все легко само порубилось и упростилось. А если не выходит каменный цветок? Нет, разумеется, можно засунуть все как есть, и посчитать в лоб. Даже если сделать это в целых числах будет весьма жирная процедура, а уж про плавающую точку я и не заикаюсь. Процессорного времени оно схавает тоже будь здоров, даже не сомневайся.

Что делать? А тут на помощь опять придет злой матан в лице аппроксимации кусочно-линейными функциями. За злой терминологией скрывается, на самом деле, примитивнейшая вещь:

Мы любую кривулину бьем на отрезки и отрисовываем ее линейными функциями. А дальше, в зависимости от величины входного значения, считаем по нужному нам отрезку.

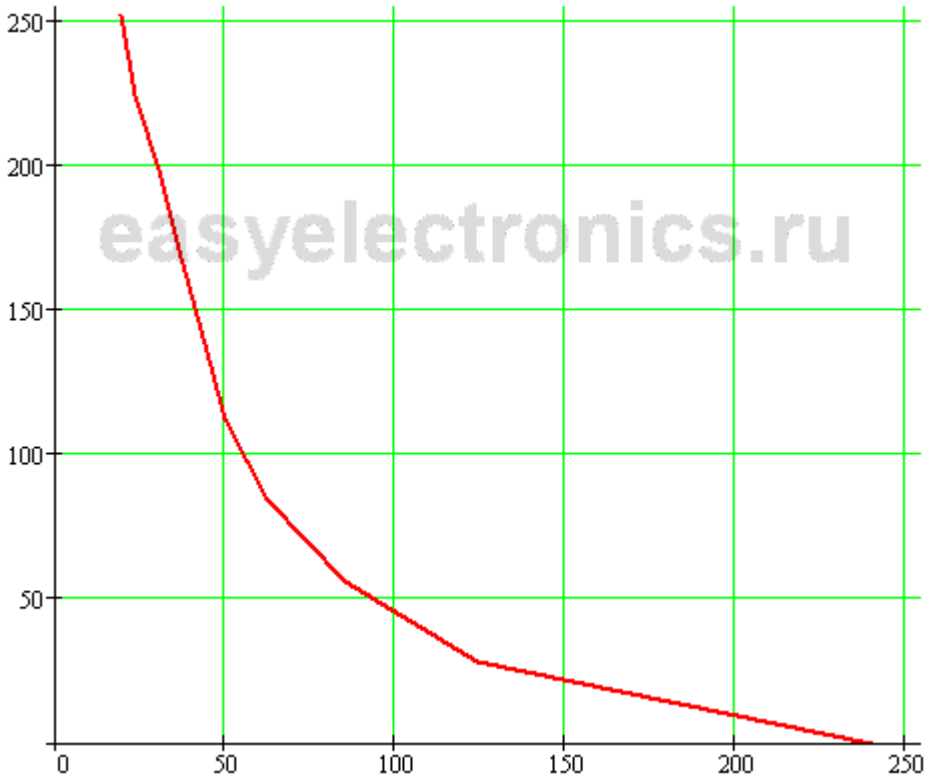
Возьмем опять наш злосчастный график характеристики оптического датчика. Я оставил от него только активную часть, ту что нам нужна на практике.



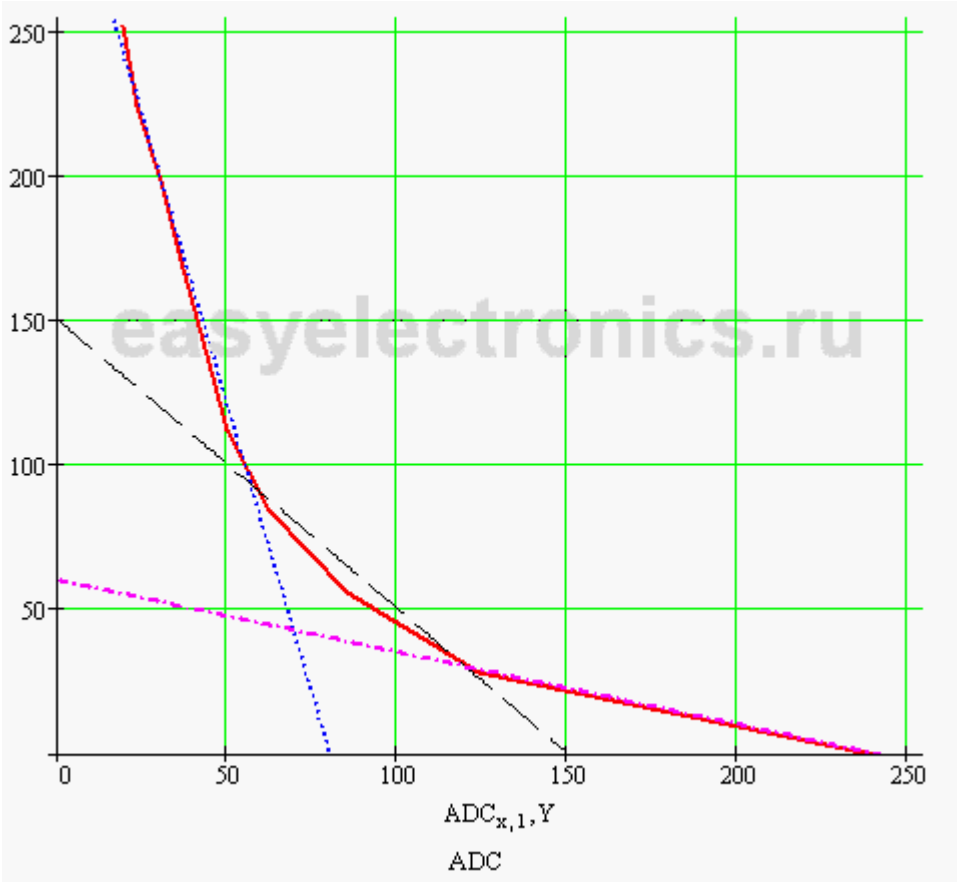
Я отрисовал его по точкам и забросил в маткад, взяв только активные области.
Вот только не понравилось мне то, что входное значение меняется от 0 до 255, а выходное от 0 до 30. Да, на выходе сантиметры, но я бы предпочел полный размах от 0 до 255. Зачем? А шумы фильтровать проще будет.

А еще повернул так, чтобы по оси X были данные из АЦП, а по оси У расстояние в наших попугаях (1 попугай = 1/7см).

Итак, возьмем и отмасштабируем нашу кривулину до 256, просто помножив на 7 с копейками.



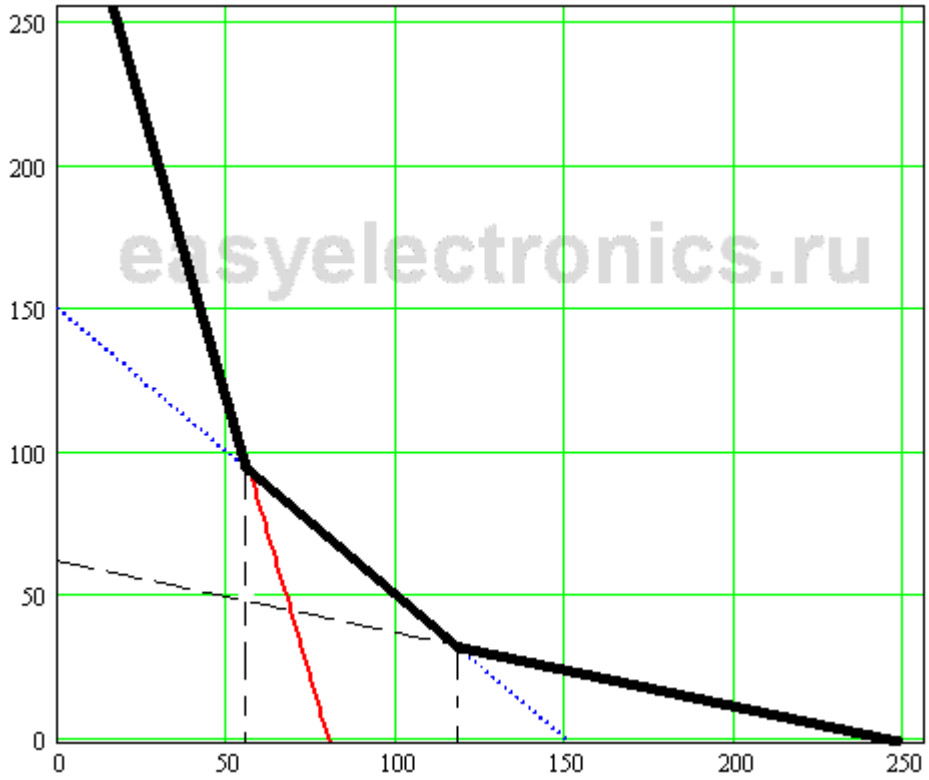
Теперь на глаз прикинем наиболее линейные участки и наложим три функции. Подбирая коэффициенты наложим их максимально точно на нашу кривую. Чем больше линейных участков тем больше памяти будет занимать наша обработка, но тем точней будет результат. Однако, скорость обработки от этого почти не зависит, ведь сколько бы у нас не было линейных участков, в каждый конкретный прогон обрабатывается только один!



Я взял три куска, описав их следующими функциями:

- X = -4Y + 320 при Y от 0 до 56
- X = -Y + 150 при Y от 56 до 118
- X = -0.25Y +60 при Y от 118 до 250

Коэффициенты я старался подобрать такие чтобы прямая не только максимально плотно легла на исходную кривую, но и чтобы они были как можно ближе к степени двойки. Чтобы можно было делить и умножать сдвигами. Если не удается подобрать степень двойки, то можно близкие к ней. Например, Y*3 = (Y<<1)+Y
Ну и в таком духе.



Вот, значит, моя спрямленная кривая.

А теперь, собственно, код:

```
1  u08 LinearAPPROX(u08 input)
2  {
3  // Too far
4  if (input<21) return 255;
5
6  // Line1 X=63+255-4Y
7  if(input<56)
8          {
9          input <<=2;                    //4*Y
10         return (63+(255-input));
11         }
12
13 // Line2 X=150-Y
14 if (input<118)
15         {
16         return (150-input);
17         }
18
19 // Line3 X=63-0.25Y
20 if(input<250)
21         {
22         input >>=2;                    //0.25*Y
```

```
23          return (63-input);
24          }
25
26 // Too close
27 return 0;
28 }
```

Заняло около 90 байт флеша, а выполняется тактов за 15-20 по самой длинной ветке. И еще один такой немаловажный момент. Обратите внимание на функцию **X = -4Y + 320** и на то как я ее представил в программе: **X=65+255-4Y**. Зачем я разбил число 320 на два куска? Да дело в том, что моя выходная величина не вылазит за предел беззнакового восьмиразрядного — это 255. А вот число 320, требуемое по уравнению, в этот лимит не влазит. И тут либо компилятор окажется умным и сам догадается, либо что нибудь куда-нибудь переполнится, что вероятней. Хорошо если Warning даст. А если не даст? Поэтому я решил подстелить соломки, сразу указав в какой последовательности и с числами какого размера оперировать, чтобы не было переполнений. Да и лишней движухи на уровне кода меньше будет.

На чистом ассемблере вообще загляденье.

```
1  ; Input:
2  ; OSRG = R17 - input data
3  ; ACC = R16
4  ; Output
5  ; OSRG - output data
6  LineAPPROX:     MOV     ACC,OSRG        ; OSRG = ACC для удобства операций
7
8                  CPI     OSRG,20         ;if(OSRG<20) слишком далеко
9                  BRCS    TooFar
10
11                 CPI     OSRG,56         ;if(OSRG<56) далеко
12                 BRCS    Far
13
14                 CPI     OSRG,118
15                 BRCS    Near            ;if(OSRG<118) средняя дистанция
16
17                 CPI     OSRG,250        ;if(OSRG<250) близко
18                 BRCS    Close
19
20 TooClose:       LDI     OSRG,0          ; совсем близко, сразу 0
21                 RET                     ; Выходим с результатом в OSRG
22
23 ; X = 63-0.25Y
24 Close:          LSR     ACC             ;0.25Y сдвигом делим на 4
25                 LSR     ACC
26
27                 LDI     OSRG,63         ;63 - (0.25Y) вычитаем
28                 SUB     OSRG,ACC
29                 RET                     ; Выходим с результатом в OSRG
30
31 ; X= 150 - Y
32 Near:           LDI     OSRG,150
33                 SUB     OSRG,ACC        ; 150-Y Не зря мы в ACC скопировали
34                 RET                     ; Выходим с результатом в OSRG
35
36 ; X = 63+255-4Y
37 Far:            LSL     OSRG            ; Y*4 сдвигом умножаем на 4
38                 LSL     OSRG
39
40                 COM     OSRG            ; 255-(Y*4)  вычли
41                 SUBI    OSRG,-63        ; 63+(255-(Y*4)) сложили
42                 RET                     ; Выходим с результатом в OSRG
43
44 TooFar: LDI     OSRG,255                ; Совсем далеко? Ну и думать нечего!
45                 RET                     ; Выходим с результатом в OSRG
```

Итого, 52 байта на весь экшн. Причем на самую длинную ветвь отводится всего 10 тактов. Красота! Можно даже в прерывании обрабатывать! :)

Да, к вопросу о том, зачем мне понадобилось растягивать диапазон до 255 попугаев. Дело в том, что несколько повышается точность выхода, но правда увеличивается уровень болтанки сигнала. Т.к. ИК порой выдает много мусора, особенно на дальних расстояниях, когда у него вытягивается характеристика и малейшие колебания превращаются в мощный расколбас выходного значения.

Так вот, когда у нас диапазон большой, то можно нажрать этой каши побольше, потом усреднить и сразу же поделить на попугаев (в том же сдвиге). На выходе, по идее, получим очень стабильные и красивые сантиметры. Впрочем, это только первичный экспериментальный прогон. Я толком еще не тестировал, но первые опыты уже дали более менее адекватные результаты.

Создание Bootloader’a

Введение
На написание данной статьи меня сподвигло практически полное отсутствие какой либо вменяемой информации по теме бутлоадеров на русском языке, и конкретно для чипов основанных на архитектуре AVR.

В общем то DI как то писал о вкусностях этих тулз для пользователей будь то мобила, либо девайс в труднодоступном месте, но процесс работы самого кода подробно не был рассмотрен.

И в один прекрасный день мне на работе дали партийное задание — разработать систему позволяющую дистанционно обновлять прошивку кое-каких устройств, сами железки стоят под взрывозащитными кожухами в шахтах на значительной глубине. Лазить туда и разбирать каждый девайс чтобы воткнуть шлейф ISP понятное дело не самая лучшая идея, однако устройства соединены интерфейсом RS485 это позволяет использовать бутлоадер в проекте.

Конечно можно взять один из OVER чем 9000 готовых бутлоадеров на Сях и доработать напильником, переделать под задачу, но мне давно было интересно разобраться в теме самопрошивки МК. И, думаю, не только мне, поэтому вооружившись даташитом и найдя скудную документацию на утилиту AVRprog я сел за AVR Studio изобетать велосипед — писать свой загрузчик. Естественно на асме (под 8ми битки только на асме пишу).

Так, для разгрева, разработаем проект бутлоадера с прошивкой по RS232 и поддержкой протокола AVRprog v1.4, а дальше можно его заточить хоть под I2C или SPI, RS485 и т. д.

WARNING
Для того чтобы полностью вкурить о чём тут говорится — рекомендуется к прочтению статья DI HALT о использовании бутлоадеров. [1]

Поехали!
Итак, первое, что нам надо сделать, кроме создания нового прожекта, это в меню AVR Studio выбрать опцию отладки бутлоадера, чтобы студия стартовала с адреса первой инструкции бутлоадера, а не 0×0000 для этого (Debug->AVR Simulation Options) выставляем флажок Enable boot reset и стартовый адрес на 0x1E00 – наш загрузчик будет занимать 512 команд.

Теперь накидаем простейший исходник – скажем чтоб выплёвывал на UART 9600/8/n/1 пару символов, это будет юзерское приложение (здесь и далее огрызки кода, подробней в полном исходнике из архива).

```
1  Reset:
2
3  ;Тут код инициализации
4  …
5  main:
6          ldi     R16, 0x55
7          rcall   SendUART
8          ldi     R16, 0xAA
9          rcall   SendUART
```

```
10          rcall   Delay
11          jmp     main
```

Далее начнём писать сам лоадер. Создаём новый модуль пропишем простенькую функцию инициализации UART'а на скорость 19200 (тут надо прописать отдельную инициализацию УАРТа отправки и приёма – надеюсь помните, что если вы используете какую-либо уарт либу из основной области памяти – она будет заблокирована в процессе прошивки и затёрта). Так ну и незабываем про адрес старта, и условие входа в бут загрузчик, стартовый код будет выглядеть примерно так:

```
1  .org 0x1E00 ; Для бутака отвели 512 слов
2  BootLoader:
3           ldi     R16, low(RAMEND) ; Никуда без стека
4           out     SPL, R16
5           ldi     R16, high(RAMEND)
6           out     SPH, R16
7
8           ldi     R16, 0xFF
9           out     DDRC, R16       ; Тут диоды у нас
10          cbi     DDRC, 4         ; А тут кнопарь
11          cbi     PORTC, 4
12
13          sbic    PINC, 4         ; Собсно и есть стартовый кондишн – если отпущена кнопка
14 бежим в основную прогу
15          jmp     Reset
16
17          LED3_ON                 ; Макрос включения светодиода «Режим программирования»
            rcall   InitUART
```

Далее идёт цикл с ожиданием байтов от AVRprog и их обработкой – довльно рутинный процесс, но поясню на примере запросов процедур идентификации – без неё AVRprog не подхватит ваш чип:

```
   Wait4Command: ; Главный цикл – тут получаем команды от AVRprog
1           rcall ReadUART
2
3  ; Тут всё просто – смотрим что в буфере R16 и идём на обработчик
4           cpi     R16, 'S'        ; Байтик приветствия – надо на него ответить 7и символьным
5                                   ; идентификатором начинающимся на "AVRxxxx" – у меня
6  например – "AVRm162"
7           breq    Greething
8
9           cpi     R16, 'a'        ; Запрос автоинкремента
10          breq    AutoInc
11
12          cpi     R16, 't'        ; Запрос типа устройства
13          breq    DeviceType
14
15          cpi     R16, 'V'        ; Запрос версии софта – у меня 0.1
16          breq    SoftwareVersion
17
18          cpi     R16, 'b'        ; Запрос поддержки буферизации – отвечаем, что
19 поддерживаем буфер
20                                  ;размером в одну страницу
21          breq    SetBufferInfo
22
            cpi R16, 's'
```

Далее по коду запросы на программирование, но о них позже, а пока что рассмотрим обработчики для этих событий на примере нескольких:

```
1  AutoInc:
2           OutUART 'y'             ;  OutUART - макрос вывода байта в UART
```

```
3         rjmp    Wait4Command
4
5  SetupLED:
6         rcall   ReadUART
7         OutUART 0xD             ; На некоторые команды надо отвечать 0xD.
8
9         rjmp    Wait4Command
10
11 SoftwareVersion:
12        OutUART '0'
13        OutUART '1'
14        rjmp    Wait4Command
15
16 GetSignature:
17        OutUART 0x1E            ; байтики сигнатуры
18        OutUART 0x94            ;
19        OutUART 0x04            ;
20
21        rjmp    Wait4Command
22
23 GetProgramerType:
24        OutUART 'S'             ; предствляемся последовательным прошивальщиком =)
25        rjmp    Wait4Command
26
27 SetBufferInfo:
28        OutUART 'Y'                     ; Ответ на запрос размера буфера (в байтах!) -
29 одна страница
30        OutUART high(PAGESIZEB)         ; PAGESIZEB= PAGESIZE*2;
31        OutUART low(PAGESIZEB)
          rjmp    Wait4Command
```

Ну чтож вот мы и подошли к самому интересному – работа с флешем.
Запросы на работу с памятью выглядят так:

```
1         cpi     R16, 'A'
2         breq    SetAddress              ; Установка адреса страницы для чтения/записи
3
4         cpi     R16, 'g'
5         breq    ReadBlock               ; Запрос на буферизированное чтение сектора
6
7         cpi     R16, 'B'
8         breq    BufferedWrite           ; Буферизированная запись
9
10        cpi     R16, 'e'
11        breq    EraseChip               ; Очистить чип
```

Теперь рассмотрим обработчики на это дело. Чтобы понять смысл следующего кода надо представлять себе принцип работы команды SPM – запись в память программ – я объясню как это работает на примере очистки чипа – дальше всё просто как 2 рубля.

```
1  EraseChip:
2         ldi     R18, 112                ;  количество страниц на очистку – зависит от чипа
3         clr     ZH
4         clr     ZL
5
6  doErase:
7         ldi R17, (1<<PGERS) | (1<<SPMEN)      ; в R17 передаётся параметр в регистр SPMCR
8  ; SPMEN – разрешает вызов команды SPM в следующих 4х тактах; PGERS – команда на очистку
9  ;страницы флеша
10
11        rcall   SPMnWAIT
12        ldi     R17, (1<<RWWSRE) | (1<<SPMEN)  ; ре-инициализация страницы
```

```
13        rcall SPMnWAIT
14
15        ldi     R19, PAGESIZE          ; инкремент указателя на страницу
16        add     ZL, R19
17        clr     R19
18        adc     ZH, R19
19        dec     R18
20        brne    doErase
21        OutUART 0xD
22        rjmp    Wait4Command
```

Сама процедура вызова SPM (ВХОД: R17 – SPMCR; ZH:ZL – указатель на страницу флеша для выполнения операций R0:R1 – непосредственное слово для записи в буфер флеша.

```
1 SPMnWAIT:
2         out     SPMCR, R17
3         spm                            ; Очищаем/пишем во флеш – зависит от параметра
4 SPMCR
5
6  WaitSPM:
7         in      R16, SPMCR
8         sbrc    R16, SPMEN
9         rjmp    WaitSPM
          ret
```

Формат регистра SPMCR

- Bit 7 – SPMIE: SPM Interrupt Enable – разрешение прерывания по окончании записи во флеш
- Bit 6 – RWWSB: Read-While-Write Section Busy – проверка занятости секции
- Bit 5 – Res: Reserved Bit
- Bit 4 – RWWSRE: Read-While-Write Section Read Enable — реинициализация страницы после записи/очистки
- Bit 3 – BLBSET: Boot Lock Bit Set – операции над фьюзами
- Bit 2 – PGWRT: Page Write – запись странцы
- Bit 1 – PGERS: Page Erase — очистка
- Bit 0 – SPMEN: Store Program Memory Enable – разрешение на исполнение SPM в следующие 4 такта.

Итак, подведём небольшой итог:

- Запись выполняется естественно командой SPM, чтение LPM
- Параметры: указатель — ZH:ZL- куда пишем (либо что очищаем), R0:R1 – слово которое пишем по (Z)
- Регистр SPMCR разрешает и определяет поведение команды SPM

Теперь внимательно рассмотрим процедуру записи флеша:
в YH:YL – указатель на начало оперы 0×100
в ZH:ZL – указатель на страницу флеша

```
1  WritePage2Flash:
2         ldi     R24, low(PAGESIZEB)    ; счётчик
3         ldi     R25, high(PAGESIZEB)
4
5  Write2FlashBuffer:
6         ld      R0, Y+
7         ld      R1, Y+
8         ldi     R17, (1<<SPMEN)        ; Тут будьте внимательны байты пишутся в буфер
9  страниц
10                                       ; флеша - в SPMCR устанавливаем только SPMEN!!!
11        rcall   SPMnWAIT
12        adiw    ZH:ZL, 2               ; указатель на текущий адрес записи
13        sbiw    R25:R24, 2
14        brne    Write2FlashBuffer
```

```
15
16 ; А ТЕПЕРЬ МОМЕНТ ИСТИННЫ – ЗАПИСЬ СТРАНИЦЫ ВО ФЛЕШ
17
18         subi    ZL, low(PAGESIZEB)                    ; восстановление указателя
19         sbci    ZH, high(PAGESIZEB)
20         ldi     R17, (1<<PGWRT)|(1<<SPMEN)            ; Даём команду на запись страницы в
21 один приём.
22         rcall   SPMnWAIT
23
24         ldi     R17, (1<<RWWSRE) | (1<<SPMEN)  ; Ре-инициализируем страницу
           rcall   SPMnWAIT
           ret
```

Вот так всё на самом деле просто выглядит!
Ну и часть обработчика вызывающего этот код:

```
1  …
2  тут очистка памяти и приём одной страницы через UART
3  …
4  ; вызываем процедуру записи страницы
5          ldi     YH, high(SRAM_START)
6          ldi     YL, low(SRAM_START)    ; Буфер в начале RAM
7          rcall   WritePage2Flash
8
9  ; Ну незабываем делать автоинкремент страницы – обязательно если при
10 ; инициализации ответили 'y' на 'A' запрос.
11
12         ldi     R19, PAGESIZE
13         add     ZL, R19
14         clr     R19
15         adc     ZH, R19
16
17         OutUART 0xD                   ; говорим AVRprog что записали страницу и всё ок!
```

Так ну и рассмотрим ещё один обработчик на этот раз последний. Адресация, AVRprog иногда наставляет нас на путь истинный, говоря конкретно в какую страницу писать, и, казалось бы, тут всё просто и понятно, но как всегда подводные камни они везде…

```
1  SetAddress:
2          rcall   ReadUART
3          mov     ZH, R16
4          rcall   ReadUART
5          mov     ZL, R16         ; ну всё просто сохраняем в регистры ZH:ZL адрес
6                                  ;куда писать и не трогаем его больше, а н-нееет…
7
8          lsl     ZL              ; адрес надо преобразовать в байтовый т. е. сдвинуть на 1
9  битик влево =)
10         rol     ZH
11
           OutUART 0xD            ; рапортуем об успешной установке адреса
```

Злоключение или это ещё не конец?
Таким образом мы рассмотрели наиболее ключевые моменты. На первый взгляд процедура самозаливки кода может показаться сложной, но на самом деле всё очень просто и сводится к реализации интерфейса для программатора — ожидать и выполнять команды. Также следует стараться уделять особое внимание некоторым неочевидным моментам (подводные камни) с которыми возможно столкнуться при написании своего проекта, для этого писалась статья и откомментирован исходник.

BootExample.zip [2]

Кстати за кадром осталась работа с прерываниями в бутлоадере (которые могут быть перемещены в секцию бутлоадера, если вы конечно их используете), установке фьюз битов, ну или перезаписи самого бутлоадера самим собой (до конца не разобрался, но такое извращение судя по всему поддерживается) –- это оставляю в качестве домашнего задания читателю. А если и возникнут сложности, либо найдутся глюки в программе, (что весьма вероятно т. к. код толком не обкатан, но как говорится – отладчик всему голова) то, возможно, распишу это всё в отдельной статье.

Ну и вопросы, предложения, критика принимается на мыло exp10der (ящик на mail.ru) или в комменты.

Гринёв Роман aka Exp10der.

AVR. Учебный Курс. Инкрементальный энкодер.

Энкодер это всего лишь цифровой датчик угла поворота, не более того.

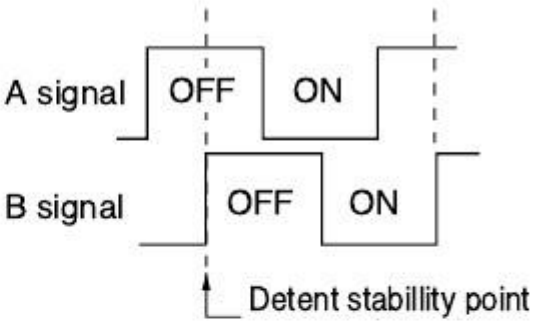
Энкодеры бывают абсолютные — сразу выдающие двоичный код угла и инкрементальные, дающие лишь указание на направление и частоту вращения, а контроллер, посчитав импульсы и зная число импульсов на оборот, сам определит положение.

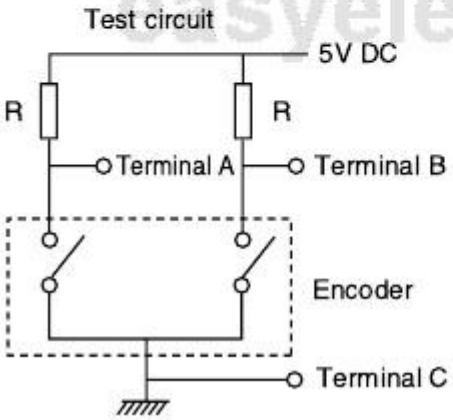
Если с абсолютным энкодером все просто, то с инкрементальным бывают сложности. Как его обрабатывать?

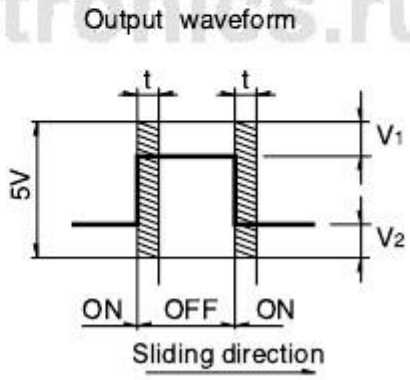
С Энкодера выходят два сигнала A и B, сдвинутых на 90 градусов по фазе, выглядит это так:

А дальше пляшем от типа энкодера. А они бывают разные.

Механический
Тебе, скорей всего, в руки попадется либо механический, либо оптический с малой дискретностью. Выдающий, в лучшем случае, пару десятков импульсов на оборот. Устроен он просто — две контактные группы замыкаются в нужном порядке в зависимости от вращения.

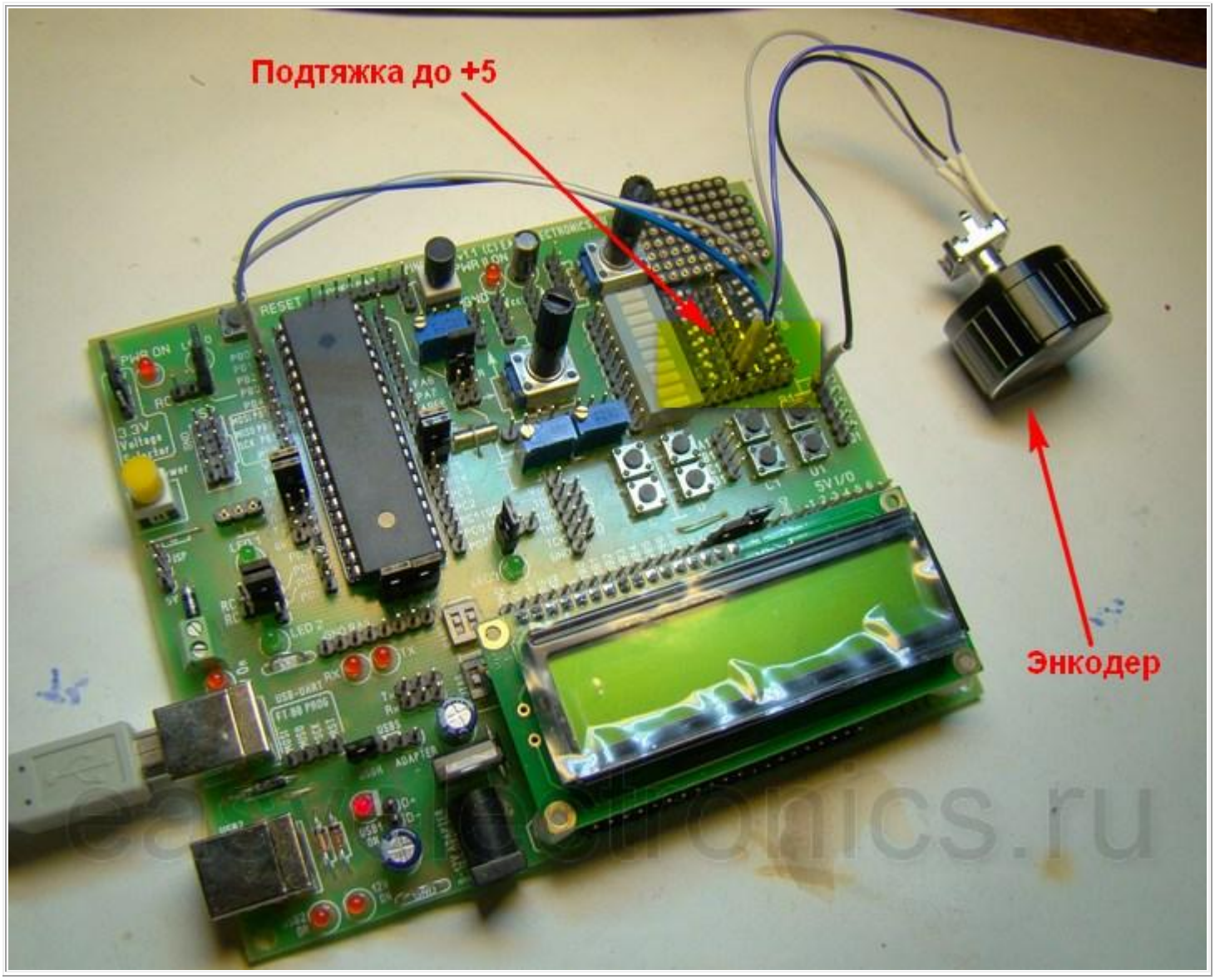






В оптическом же может быть два фонаря и два фотодиода, святящие через диск с прорезями (шариковая мышка, ага. Оно самое).

Механический подключается совсем просто центральный на землю, два крайних (каналы) на подтянутые порты. Я, для надежности, подключил внешнюю подтяжку. Благо мне на Pinboard [1] для этого только парой тумблеров щелкнуть:

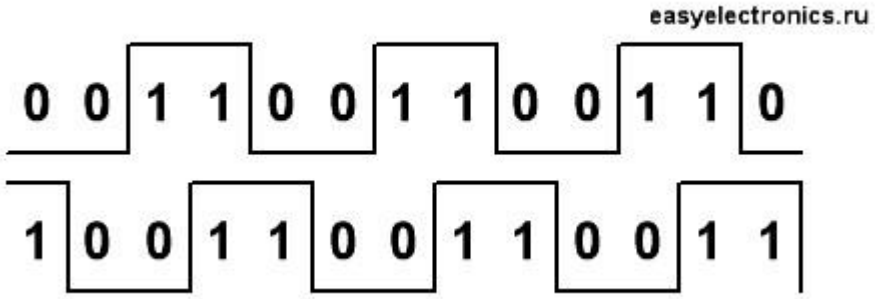


Оптический подключается в зависимости от типа оптодатчика, обычно там стоит два фотодиода [2] с общим анодом.

Обычно, все пытаются работать с ними через прерывания INT, но этот метод так себе. Проблема тут в дребезге — механические контакты, особенно после длительного пользования, начинают давать сбои и ложные импульсы в момент переключения. А прерывание на эти ложные импульсы все равно сработает и посчитает что нибудь не то.

Лучше считать не импульсы, а состояния.

Метод прост:
Подставим нули и единички, в соответствии с уровнем сигнала и запишем последовательность кода:

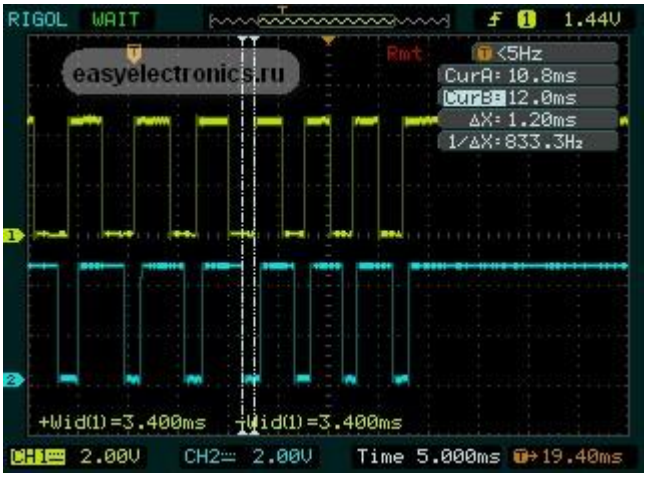


A:0 0 1 1 0 0 1 1 0 0 1 1 0
B:1 0 0 1 1 0 0 1 1 0 0 1 1

Если A и B идут на одни порт контроллера (например на A=PB0 B=PB1), то при вращении энкодера у нас возникает меняющийся код:

11 = 3
10 = 2
00 = 0
01 = 1
11 = 3

Теперь остается только циклически опрашивать наш энкодер сравнивая текущее состояние с новым и на основании этого делающего выводы о вращении. Причем частота опроса должна быть такой, чтобы не пропустить ни одного импульса. Например, мой EC12 имеет 24 импульса на оборот. Вращать его предполагается вручную и я вряд ли смогу вращать его с космической скоростью, но решил все же замерить. Подключился к осциллографу, крутнул ручку что есть мочи:



Выжал меньше килогерца. Т.е. опрашивать надо примерно 1000 раз в секунду. Можно даже реже, будет надежней в плане возможного дребезга. Сейчас, кстати, дребезга почти нет, но далеко не факт что его не будет потом, когда девайсина разболтается.

Сам опрос должен быть в виде конечного автомата. Т.е. у нас есть текущее состояние и два возможных следующих.

```
1  // Эту задачу надо запускать каждую миллисекунду.
2  // EncState глобальная переменная u08 -- предыдущее состояние энкодера
3  // EncData глобальная переменная u16 -- счетный регистр энкодера
4
5
6  void EncoderScan(void)
7  {
8  u08 New;
```

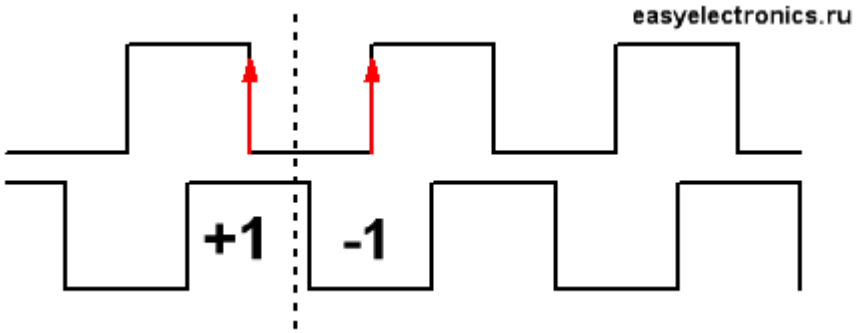
```
9
10 New = PINB & 0x03;      // Берем текущее значение
11                         // И сравниваем со старым
12
13 // Смотря в какую сторону оно поменялось -- увеличиваем
14 // Или уменьшаем счетный регистр
15
16 switch(EncState)
17         {
18         case 2:
19                 {
20                 if(New == 3) EncData++;
21                 if(New == 0) EncData--;
22                 break;
23                 }
24
25         case 0:
26                 {
27                 if(New == 2) EncData++;
28                 if(New == 1) EncData--;
29                 break;
30                 }
31         case 1:
32                 {
33                 if(New == 0) EncData++;
34                 if(New == 3) EncData--;
35                 break;
36                 }
37         case 3:
38                 {
39                 if(New == 1) EncData++;
40                 if(New == 2) EncData--;
41                 break;
42                 }
43         }
44
45 EncState = New;         // Записываем новое значение
46                                 // Предыдущего состояния
47
48 SetTimerTask(EncoderScan,1);   // Перезапускаем задачу через таймер диспетчера
49 }
```

Почему я под счетчик завел такую большую переменную? Целых два байта? Да все дело в том, что у моего энкодера, кроме импульсов есть еще тактильные щелчки. 24 импульса и 24 щелчка на оборот. А по моей логике, на один импульс приходится четыре смены состояния, т.е. полный период 3201_3201_3201 и один щелчок дает 4ре деления, что некрасиво. Поэтому я считаю до 1024, а потом делю сдвигом на четыре. Получаем на выходе один щелочок — один тик.

Скоростной опрос на прерываниях
Но это механические, с ними можно простым опросом обойтись — частота импульсов позволяет. А бывают еще и высокоскоростные энкодеры. Дающие несколько тысяч импульсов на оборот, либо работающие на приводах и вращающиеся очень быстро. Что с ними делать?

Ускорять опрос занятие тупиковое. Но нас спасает то, что у таких энкодеров, как правило, есть уже свои схемы подавления дребезгов и неопределенностей, так что на выходе у них четкий прямоугольный сигнал (правда и стоят они совершенно негуманно. От 5000р и до нескольких сотен тысяч. А что ты хотел — промышленное оборудование дешевым не бывает).

Так что без проблем можно применять прерывания. И тогда все упрощается неимоверно. Настраиваем всего одно прерывание по внешнему сигналу. Например, INT0 настраиваем так, чтобы сработка шла по восходящему фронту. И подаем на INT0 канал А.



Пунктиром показано предполагаемое положение в произвольный момент. Красные стрелки это фронты по которым сработают прерывания при движении либо в одну, либо в другую сторону.

А в обработчике прерывания INT0 щупаем вторым выводом канал B. И дальше все элементарно!

Если там высокий уровень — делаем +1, если низкий -1 нашему счетному регистру. Кода на три строчки, мне даже писать его лень.

Конечно, можно такой метод прикрутить и на механический энкодер. Но тут надо будет заблокировать прерывания INT0 на несколько миллисекунд. И НИ В КОЕМ СЛУЧАЕ нельзя делать это в обработчике.

Алгоритм прерывания с антидребезгом будет выглядеть так:

- Зашли в обработчик INT0
- Пощупали второй канал
- +1 или -1
- Запретили локально INT0
- Поставили на таймер событие разрешающее INT0 через несколько миллисекунд
- Вышли из обработчика

Сложно? Нет, не сложно. Но зачем? Проще сделать банальный опрос, как указано выше и не зависеть от выводов прерываний. Впрочем, хозяин барин.

Обработка множества инкрементальных энкодеров одновременно

Про инкрементальный энкодер и про обработку его сигналов с помощью МК уже была статья [1]. Вроде-бы ничего сложного — два бита текущего состояния, два бита предыдущего — автомат с 16 состояниями. Рассмотрим эту задачу ещё раз с позиции максимально эффективной (по скорости и размеру кода) обработки сигналов множества энкодеров одновременно.

Обозначим текущее состояние энкодера как «y1» и «y2», а предыдущее, как «x1» и «x2». Всего 4 бита — 16 состояний. Условимся, что направление «Вперёд» у нас будет от первого датчика энкодера ко второму. Запишем все возможные состояния в таблицу.

Таблица 1.

№	y2	y1	x2	x1	Вперёд	Назад	Состояние
0	0	0	0	0	0	0	Стоп
1	0	0	0	1	0	1	Назад
2	0	0	1	0	1	0	Вперёд
3	0	0	1	1	0	0	Не определено
4	0	1	0	0	1	0	Вперед
5	0	1	0	1	0	0	Стоп
6	0	1	1	0	0	1/0	Назад*
7	0	1	1	1	0	1	Назад
8	1	0	0	0	0	1	Назад

9	1	0	0	1	1/0	0	Вперёд*
A	1	0	1	0	0	0	Стоп
B	1	0	1	1	1	0	Вперёд
C	1	1	0	0	0	0	Не определено
D	1	1	0	1	1	0	Вперёд
E	1	1	1	0	0	1	Назад
F	1	1	1	1	0	0	Стоп

* — строчки 6 и 9 в таблице в принципе означают перемещение назад и вперёд соответственно, в случае если оба датчика энкодера никогда не срабатывают одновременно. Такая ситуация теоретически может иметь место если энкодер это две оптопары и колесо с отверстиями, причем размер отверстия меньше расстояния между оптопарами. На практике это встречается редко, по этому будем иметь этот случай ввиду, но учитывать не будем.

Теперь в соответствии с таблицей напишем код определяющий направление вращения энкодера. Самый простой и тем не менее достаточно эффективный вариант вариант это — упаковать все 4 бита в одну переменную и сделать switch по ней:

```
1  static uint8_t EncState=0;
2
3  static volatile uint16_t EncValue=0;
4
5  inline static void EncoderScan(void)
6  {
7          uint8_t newValue = PINC & 0x03;
8  s       uint8_t fullState = newValue | EncState << 2;
9
10         switch(fullState)
11         {
12                 case 0x2: case 0x4: case 0xB: case 0xD:
13                         EncValue ++;
14                 break;
15                 case 0x1: case 0x7: case 0x8: case 0xE:
16                         EncValue --;
17                 break;
18         }
19         EncState = newValue;
20 }
```

Тут мы воспользовались возможностью задавать несколько меток case для одного блока. Значение меток соответствуют номерам строк из нашей таблицы — очень удобно и наглядно, легко обработать и другие состояния если надо.
Теперь приступим к количественным измерениям. Для чистоты эксперимента сканирование энкодера будем помещать в обработчик прерывания, например, от таймера — там будет сразу видно сколько регистров надо сохранять. Сами функции сканирования энкодера будем делать встраиваемыми, что их тело помещалось непосредственно в обработчик прерывания. Размер будем считать от начала обработчика по reti включительно. Целевой процессор — Mega16. Компиляторы avr-gcc 4.3 и IAR C/C++ Compiler for AVR 5.50.0 [KickStart]. Во всех случаях оптимизация кода по размеру. Такты затраченные на выполнение определялись на симуляторах AvrStudio и IAR EWAVR соответственно.

Результаты для этой функции:

- gcc – 112 байт кода, время выполнения примерно 57-62 тактов.
- IAR – 116 байт кода, 62-66 тактов.

Первым указано количество тактов если состояние энкодера не изменилось, вторым — если изменилось. Количество тактов может несколько меняться в зависимости от того, по какой ветке оператора switch пошла программа, но диапазон этого изменения примерно такой.
Примерно треть времени тратится на сохранение восстановление регистров.
Вполне неплохо если нам нужен только один энкодер, но нам их надо много. При масштабировании этого подхода размер и время обработки растут практически пропорционально числу энкодеров. И если с рост размера можно ограничить написав функцию EncoderScan так, чтобы она принимала состояние и указатель на счетчик в качестве параметров, то скорость обработки от этого только упадёт. Контроллер, ведь, не только энкодеры обрабатывать

должен, у него еще работа есть. Большую часть времени у нас занимает непосредственно определение направления движения, к тому-же оно выполняется для каждого энкодера последовательно.

Немного логики
Посмотрим на задачу определения направления вращения энкодера формально:
Есть две логические функции «Вперёд» и «Назад». Они принимают 4 логических параметра и возвращают 1 в случае движения вперёд или назад соответственно. Заданны эти функции таблицей истинности. А по таблице истинности можно уже синтезировать логическое выражение. В нашем контексте задачи это означает, что мы можем упаковать все значения x1, x2, y1, y2 всех энкодеров в отдельные целочисленные переменные, и обрабатывать разом данные со стольких энкодеров, сколько бит в этих переменных. Неплохо так параллельно определить направление вращения сразу до 8/16/32 энкодеров. Изменять значения счётчиков, конечно придётся в цикле, параллельно это сделать уже не удастся.
Теперь только остаётся синтезировать это самое логическое выражение. Возьмёмся для начала за функцию «Вперёд». Найдём в нашей таблице все единичные значения этой функции:

Таблица 2.

№	y2	y1	x2	x1	Вперёд	Назад	Состояние
2	0	0	1	0	1	0	Вперёд
4	0	1	0	0	1	0	Вперед
B	1	0	1	1	1	0	Вперёд
D	1	1	0	1	1	0	Вперёд

Запишем для нашей функции логическое выражение, сразу на языке Си, чтоб не мучатся с математической нотацией:

```
Fwd =   ~x1 &  x2 &  ~y1 & ~y2 |
        ~x1 & ~x2 &   y1 & ~y2 |
         x1 &  x2 &  ~y1 &  y2 |
         x1 & ~x2 &   y1 &  y2;
```

Каждая строчка этого выражения соответствует одной строке в таблице. Если в таблице аргумент имеет значение «0», то в нашем выражении записываем его с отрицанием «~» (инверсия всех бит). Если он равен «1», то без отрицания. Например, для первой строчки, только x2 имеет единичное значение, x2 берём непосредственно, остальные аргументы с отрицанием: ~x1 & x2 & ~y1 & ~y2. Это выражение вернёт 1 только если x2 равен 1, а остальные параметры 0. Склеивая выражения для каждой строчки с помощью операции ИЛИ мы получим искомую функцию.
Но эта функция не оптимальна, её можно и нужно оптимизировать. Для этого воспользуемся законами логики.

В первых двух строчках вынесем за скобки **~x1 & ~y2**, а в последних двух — вынесем **x1 & y2**:

```
~x1 & ~y2 &  (x2 & ~y1 | ~x2 & y1)|
 x1 &  y2 &  (x2 & ~y1 | ~x2 & y1) ;
```

Выражение в скобках **(x2 & ~y1 | ~x2 & y1)** это ни что иное, как исключающее ИЛИ — **x2 ^ y1**.
Выражение ещё упростилось:

```
~x1 & ~y2 & (x2 ^ y1 )|
 x1 &  y2 & (x2 ^ y1 ) ;
```

Теперь вынесем за скобки x2 ^ y1 и получим:

```
(x2 ^ y1 ) &  (~x1  & ~y2  |  x1   &   y2 );
```

Во вторых скобках если заменить «x» на «~x», то у нас снова получается исключающее ИЛИ:

```
(x2 ^ y1 ) &  (~x1  ^  y2 );
```

Инверсия одного из аргументов исключающего ИЛИ приводит к инверсии всего выражения, значит инверсию можно вынести за скобки:

```
(x2 ^ y1 ) &  ~(x1  ^  y2 );
```

В результате получилось достаточно простое выражение (всего 4 операции) для определения вращения энкодера вперёд. Это выражение симметрично относительно индексов 1 и 2, и поменяв их местами получим выражение определяющее движение назад:

```
(x1 ^ y2 ) &  ~(x2  ^  y1 );
```

Теперь осталось только реализовать обработку нескольких энкодеров на языке Си.
Ограничимся для начала максимум 8 энкодерами, чтобы значения умещались в тип uint8_t.

```
1         // Тип переменных-счетчиков
2         typedef unsigned EncValueType;
3
4         //количество обрабатываемых энкодеров
5         enum{EncoderChannels = 8};
6
7         // Массив переменных-счетчиков
8         static volatile EncValueType EncoderValues[EncoderChannels];
9         // предыдущие состояния энкодеров
10        static uint8_t _x1, _x2;
11
12        // Определение вращения вперёд/назад
13        static inline uint8_t Detect(uint8_t x1, uint8_t x2, uint8_t y1, uint8_t y2)
14        {
15                //вот оно наше волшебное выражение
16                return (x2 ^ y1) & ~(x1 ^ y2);
17        }
18
19        //функции чтения текущего состояния энкодеров. Первый в второй датчики
20 соответственно.
21        // определим их позже
22        inline uint8_t EncRead1();
23        inline uint8_t EncRead2();
24
25        static inline void EncoderCapture()
26        {
27                // читаем текущее состояние сразу всех энкодеров
28                uint8_t y1 = EncRead1();
29                uint8_t y2 = EncRead2();
30
31                // определяем наличие движения вперёд
32                uint8_t fwd  = Detect(_x1, _x2, y1, y2);
33                // меняем индексы 1 и2 местами и определяем наличие движения назад
34                uint8_t back = Detect(_x2, _x1, y2, y1);
35
36                // сохраняем текущее состояние
37                _x1 = y1;
38                _x2 = y2;
39
40                // в цикле проходим по массиву счётчиков энкодеров
41                volatile EncValueType * ptr = EncoderValues;
42                for(uint8_t i = EncoderChannels; i; --i)
43                {
44                        if(fwd & 1)
45                                (*ptr) ++;
46                        else
47                        if(back & 1)
48                                (*ptr) --;
49                        ptr++;
50                        fwd >>= 1;
51                        back >>= 1;
52                }
```

```
53          }
54
55          // функции чтения текущего состояния энкодеров.
56          // реализуем их как душе угодно, то есть как энкодеры подключены.
57          //Нулевой бит в обоих значениях соответствует нулевому энкодеру, первый — первому
58  и т.д.
59          inline uint8_t EncRead1()
60          {
61                  return PINC;
62          }
63
64          inline uint8_t EncRead2()
65          {
                    return PIND;
            }
```

Итак, посмотрим на результаты:

avr-gcc:

- 1 энкодер 108 байт — 63-70 такта
- 2 энкодера 162 байт — 77-84 тактов
- 8 энкодеров 138 байт — 216-276 тактов

IAR:

- 1 энкодер 128 байт — 86-94 такта
- 2 энкодера 128 байт — 100-116 тактов
- 8 энкодеров 128 байт — 186-248 тактов

При использовании только одного энкодера, результат примерно сопоставимый с вариантом на базе оператора switch, даже чуть похуже. Однако, уже при обсчете двух энкодеров преимущество становится очевидным. Для восьми — оно ещё более значимо. Здесь основное время уже занимает обход массива со счетчиками и изменение их значения.

- **Файл проекта для AVRGCC и IAR** [2]

Заключение
Применение таких логических функций позволяет распараллелить вычисления для нескольких автоматов на битовом уровне. Это позволяет сократить время выполнения кода и его размер в некоторых случаях в разы. Подобный подход целесообразно применять в случае если имеется несколько однотипных устройств, управление которыми осуществляется с помощью простого автомата состояний, например, шаговыми и BLDC двигателями.

Организация древовидного меню

Почти для всех проектов на микроконтроллере с экранчиком требуется система меню. Для каких-то проектов одноуровневое, для других — многоуровневое древовидное. Памяти, как обычно, мало, поэтому хочется запихнуть все во флэш.

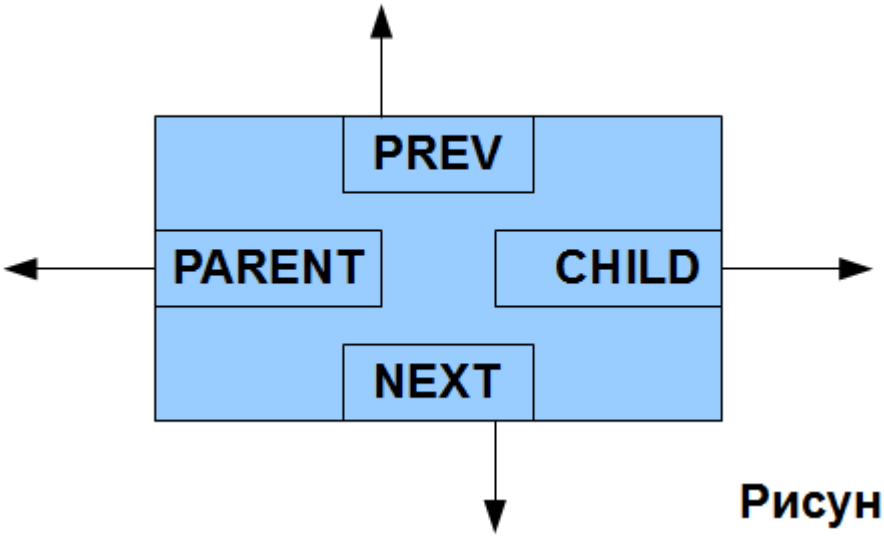


Рисунок 1

Попутно, из проекта в проект, развивалась своя псевдоОС — таймеры, события, диспетчеры.
Я ее полностью писал на си, оптимизацией и вылизыванием еще не занимался.

Перебирая разные системы, наткнулся на MicroMenu [1]:

Попробуем разобрать ее на части и прикрутить к системе.

Структура данных:
Меню организовано в виде четырехсвязного списка. Каждый элемент меню (пункт меню) ссылается на предыдущего и последующего элемента, также ссылается на своего родителя (пункт меню предыдущего уровня) и потомка (пункт подменю). Если это первый пункт, то предыдущего элемента у него нет, соответствующая ссылка пустая.

Изобразим это на рисунке:

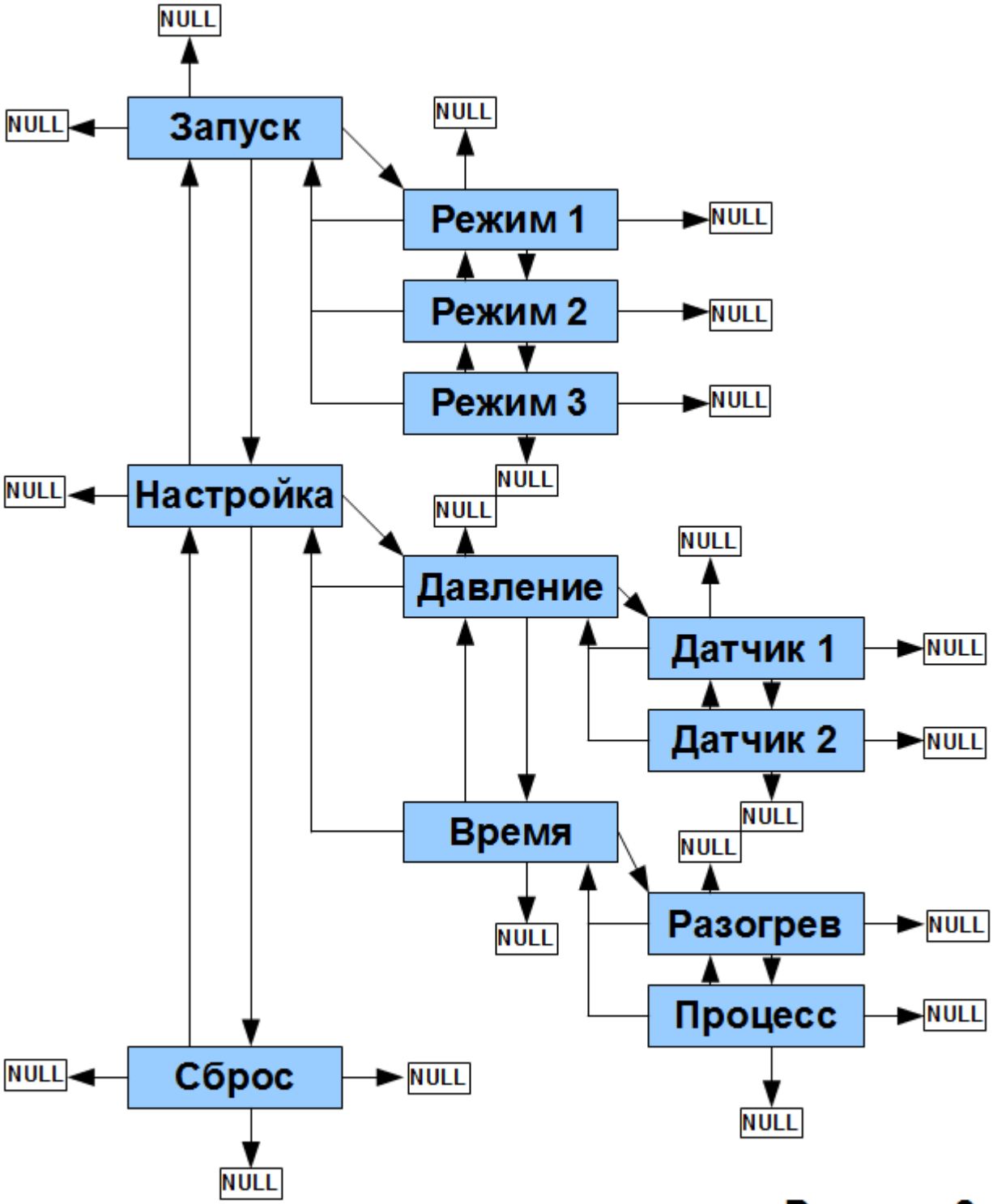


Рисунок 2

Для чего такая избыточность? По сути, с текущим пунктом меню можно сделать четыре вещи:

- Перейти вверх или вниз (предыдущий или следующий пункт)
- Вернуться в родительское меню (если есть)
- Выбрать текущий пункт. При этом мы или переходим в подменю (ели оно есть), или выполняется команда, прикрепленная к этому пункту меню.

При наличии джойстика (или четырех кнопок «крестом») эти действия как раз вешаются на свою кнопку.

Соответственно, все эти действия отражают четыре указателя. В оригинальной системе указатель на потомка обозван SIBLING, но я считаю это идеологически неверным. Sibling – это родственник того же уровня. Брат или сестра. Но никак не потомок. Поэтому мы будем использовать идеологически выверенное CHILD.

Итак, описание структуры пункта меню:

```
typedef struct PROGMEM{
        void       *Next;
        void       *Previous;
        void       *Parent;
        void       *Child;
        uint8_t     Select;
        const char  Text[];
} menuItem;
```

Добавлен байт Select – это код команды, привязанный к текущему пункту. Если у данного пункта есть подменю, код нулевой. Также есть поле Text. Капитан Очевидность подсказывает, что это, собственно, текст пункта меню. По расходам памяти — на каждый пункт меню расходуется 9 байт плюс длина текстовой части. И это все — кладется во флеш.

Самое полезное, почерпнутое у MicroMenu – набор дефайнов для быстрого и удобного определения меню.

```
#define MAKE_MENU(Name, Next, Previous, Parent, Child, Select, Text) \
        extern menuItem Next;     \
        extern menuItem Previous; \
        extern menuItem Parent;   \
        extern menuItem Child;  \
        menuItem Name = {(void*)&Next, (void*)&Previous, (void*)&Parent, (void*)&Child,
(uint8_t)Select, { Text }}
```

В чем пафос такой конструкции? Для того, чтобы определить текущий элемент, нам надо указать ссылку на следующий, еще не известный компилятору. Поэтому этот дефайн создает заведомо избыточное количество описаний extern. Это означает, что такой идентификатор будет где-то описан, не обязательно в этом же файле. В качестве бонуса это позволит растащить меню по нескольким файлам, если вдруг возникнет такое неудовлетворенное желание.

Теперь самое интересное: описание структуры меню, как на рисунке.

```
// для начала — пустой элемент. Который NULL на рисунке
#define NULL_ENTRY Null_Menu
menuItem        Null_Menu = {(void*)0, (void*)0, (void*)0, (void*)0, 0, {0x00}};

enum {
    MENU_CANCEL=1,
    MENU_RESET,
    MENU_MODE1,
    MENU_MODE2,
    MENU_MODE3,
    MENU_SENS1,
    MENU_SENS2,
};

//                NEXT,       PREVIOUS     PARENT,     CHILD
MAKE_MENU(m_s1i1,  m_s1i2,     NULL_ENTRY,  NULL_ENTRY, m_s2i1,        0, "Запуск");
MAKE_MENU(m_s1i2,  m_s1i3,     m_s1i1,      NULL_ENTRY, m_s3i1,        0, "Настройка");
MAKE_MENU(m_s1i3,  NULL_ENTRY,m_s1i2,       NULL_ENTRY, NULL_ENTRY,    MENU_RESET,
"Сброс");

// подменю Запуск
MAKE_MENU(m_s2i1,  m_s2i2,     NULL_ENTRY,  m_s1i1,     NULL_ENTRY,    MENU_MODE1, "Режим
1");
```

```
24 MAKE_MENU(m_s2i2,   m_s2i3,    m_s2i1,      m_s1i1,     NULL_ENTRY,   MENU_MODE2, "Режим
25 2");
26 MAKE_MENU(m_s2i3,   NULL_ENTRY,m_s2i2,      m_s1i1,     NULL_ENTRY,   MENU_MODE3, "Режим
27 3");
28
29 // подменю Настройка
30 MAKE_MENU(m_s3i1,   m_s3i2,    NULL_ENTRY,  m_s1i2,     m_s4i1,       0, "Давление");
31 MAKE_MENU(m_s3i2,   NULL_ENTRY,m_s3i1,      m_s1i2,     m_s5i1,       0, "Время");
32
33 // подменю Давление
34 MAKE_MENU(m_s4i1,   m_s4i2,    NULL_ENTRY,  m_s3i1,     NULL_ENTRY,   MENU_SENS1, "Датчик
35 1");
   MAKE_MENU(m_s4i2,   NULL_ENTRY,m_s4i1,      m_s3i1,     NULL_ENTRY,   MENU_SENS2, "Датчик
   2");

   // подменю Время
   MAKE_MENU(m_s5i1,   m_s5i2,    NULL_ENTRY,  m_s3i2,     NULL_ENTRY,   MENU_WARM,
   "Разогрев");
   MAKE_MENU(m_s5i2,   NULL_ENTRY,m_s5i1,      m_s3i2,     NULL_ENTRY,   MENU_PROCESS,
   "Процесс");
```

Готово!

Естественно, пункты меню можно описывать и вперемешку, в порядке обхода дерева. Типа такого:

```
  MAKE_MENU(m_s1i1,   m_s1i2,    NULL_ENTRY,  NULL_ENTRY, m_s2i1,       0, "Запуск");
1     // подменю Запуск
2     MAKE_MENU(m_s2i1,   m_s2i2,    NULL_ENTRY,  m_s1i1,     NULL_ENTRY,   MENU_MODE1,
3  "Режим 1");
4     MAKE_MENU(m_s2i2,   m_s2i3,    m_s2i1,      m_s1i1,     NULL_ENTRY,   MENU_MODE2,
5  "Режим 2");
6     MAKE_MENU(m_s2i3,   NULL_ENTRY,m_s2i2,      m_s1i1,     NULL_ENTRY,   MENU_MODE3,
   "Режим 3");
  MAKE_MENU(m_s1i2,   m_s1i3,    m_s1i1,      NULL_ENTRY, m_s3i1,       0, "Настройка");
```

Можно даже пойти дальше — строить меню в какой-нибудь визуальной среде, а потом автоматически генерировать такой список. Но это на потом.

Плюсы и минусы такой организации. Минус — явная избыточность. Плюс — возможность быстро редактировать меню — вставить новый пункт, поменять местами, удалить. Изменяются только соседние элементы меню, без тотальной перенумерации. Мне этот плюс перевесил все остальные минусы.

Опять же бонус — можно организовать несколько не связанных друг с другом деревьев меню. Главное не потерять точку входа.

Дальше. Как ходить по меню. Автор предлагает несколько дефайнов. Я их сохранил, хотя можно и без них обойтись.

```
1  #define PREVIOUS   ((menuItem*)pgm_read_word(&selectedMenuItem->Previous))
2  #define NEXT       ((menuItem*)pgm_read_word(&selectedMenuItem->Next))
3  #define PARENT     ((menuItem*)pgm_read_word(&selectedMenuItem->Parent))
4  #define CHILD      ((menuItem*)pgm_read_word(&selectedMenuItem->Child))
5  #define SELECT     (pgm_read_byte(&selectedMenuItem->Select))
6
7  menuItem* selectedMenuItem; // текущий пункт меню
8
9  void menuChange(menuItem* NewMenu)
10 {
11         if ((void*)NewMenu == (void*)&NULL_ENTRY)
12           return;
13
```

```
14          selectedMenuItem = NewMenu;
15 }
```

Вроде должно быть понятно. Выполняется проверка, если есть куда переходить, то переходим. Иначе — не переходим. Вызывается эта процедура таким образом:

```
1 menuChange(PREVIOUS);
```

Далее, процедура реакции на нажатие клавиш (в качестве параметра передается код нажатой клавиши):

```
1  uint8_t keyMenu(msg_par par) {
2          switch (par) {
3          case 0: {
4                  return 1;
5          }
6          case KEY_UP: {
7                  menuChange(PREVIOUS);
8                  break;
9          }
10         case KEY_DOWN: {
11                 menuChange(NEXT);
12                 break;
13         }
14         case KEY_RIGHT:
15                 ;
16         case KEY_OK:
17                 { // выбор пункта
18                         uint8_t sel;
19                         sel = SELECT;
20                         if (sel != 0) {
21                                 sendMessage(MSG_MENU_SELECT, sel);
22
23                                 killHandler(MSG_KEY_PRESS, &keyMenu);
24                                 killHandler(MSG_DISP_REFRESH, &dispMenu);
25
26                                 return (1);
27                         } else {
28                                 menuChange(CHILD);
29                         }
30                         break;
31                 }
32         case KEY_LEFT: { // отмена выбора (возврат)
33                 menuChange(PARENT);
34         }
35         }
36         dispMenu(0);
37         return (1);
38 }
```

Процедура отрисовки меню. Зависит от выбранного экранчика, а также от используемой анимации при выборе. Например у меня экранчик 128x64 точки, текущий пункт меню всегда по середине экрана, сверху и снизу выводятся два предыдущих и два последующих элемента (если есть). Отрисовка вызывается после каждого нажатися на кнопку и по таймеру, два раза в секунду. Мало ли, может изменится что.
В шапке можно выводить текст родителя, чтобы знать, где находимся. Можно двигать курсор по пунктам, а не пункты прокручивать. На вкус и цвет все фломастеры разные.

```
1  char* menuText(int8_t menuShift)
2  {
3          int8_t i;
4          menuItem* tempMenu;
5
```

```
6           if ((void*)selectedMenuItem == (void*)&NULL_ENTRY)
7             return strNULL;
8
9           i = menuShift;
10          tempMenu = selectedMenuItem;
11          if (i>0) {
12                  while( i!=0 ) {
13                          if ((void*)tempMenu != (void*)&NULL_ENTRY) {
14                                  tempMenu = (menuItem*)pgm_read_word(&tempMenu->Next);
15                          }
16                          i--;
17                  }
18          } else {
19                  while( i!=0 ) {
20                          if ((void*)tempMenu != (void*)&NULL_ENTRY) {
21                                  tempMenu = (menuItem*)pgm_read_word(&tempMenu->Previous);
22                          }
23                          i++;
24                  }
25          }
26
27          if ((void*)tempMenu == (void*)&NULL_ENTRY) {
28                  return strNULL;
29          } else {
30                  return ((char *)tempMenu->Text);
31          }
32  }
33
34  unsigned char dispMenu(msg_par par) {
35          char buf[25];
36
37          LCD_clear();
38
39          sprintf_P(buf, PSTR(">%-20S"), menuText(0));
40          LCD_putsXY(2, 28, buf);
41
42          sprintf_P(buf, PSTR("%-20S"), menuText(-2));
43          LCD_putsXY(2, 8, buf);
44
45          sprintf_P(buf, PSTR("%-20S"), menuText(-1));
46          LCD_putsXY(5, 17, buf);
47
48
49          sprintf_P(buf, PSTR("%-20S"), menuText(1));
50          LCD_putsXY(5, 39, buf);
51
52          sprintf_P(buf, PSTR("%-20S"), menuText(2));
53          LCD_putsXY(2, 48, buf);
54
55          LCD_drawLine(0, 26, 127, 26, 1);
56          LCD_drawLine(0, 34, 127, 34, 1);
57
58          return (1);
59  }
```

И последний штрих — инициализация меню:

```
1  void startMenu(void) {
2          selectedMenuItem = (menuItem*)&m_l1i1;
3
4          dispMenu(0);
5          setHandler(MSG_KEY_PRESS, &keyMenu);
6          setHandler(MSG_DISP_REFRESH, &dispMenu);
```

```
7 }
```

Для начала хватит. В продолжении — сделать несколько меню, сделать процедуру работы с меню реентерабельной, забабахать модель в протеусе.

Краткое описание того, что делает процедура setHandler — она привязывает обработчик к событию. В данном случае, при возникновении события MSG_KEY_PRESS вызовется функция keyMenu для обработки этого события.

Для демонстрации системы меню, описанной в предыдущем посте, собрал модель в протеусе. На базе двухстрочного LCD-индикатора, контроллераatmega32 и пяти кнопок (влево-вправо-вверх-вниз и выбор). В своих схемах использую джойстики от мобилок, они тоже пятипозиционные. Также воткнул три светодиода, чтобы хоть как-то реагировать на выбор пунктов меню.

Поскольку на экране у нас всего две строчки, решил отображение меню сделать горизонтальным. В верхней строчке отображается родительский пункт меню (или просто «Меню:», если мы на верхнем уровне), во второй строчке — текущий пункт меню. Клавишами влево-вправо выбираем предыдущий-следующий. Клавишей вверх — возвращаемся в родительское меню, клавиша вниз (или ОК) — заходим в выбранный пункт.

Обработка меню:

- при выборе пунктов Start/Mode 1, Start/Mode 2, Start/Mode 3 загорается соответствующий светодиод
- при выборе пункта Reset — все светодиоды гаснут
- после выбора любого конечного пункта, возвращаемся обратно в корень меню.

Некоторые модификации, связанные с моделированием:

- заменил весь текст на английский, потому что в оригинале модель экранчика не поддерживает русский язык. Да, я знаю, про замену dll, но не у всех она есть, а для просмотра — пойдет.
- моя боевая библиотека работы с LCD почему-то отказалась работать с моделью. Поэтому взял какую-то первую попавшуюся из древнего проекта.
- отключил автоповтор кнопок (на модели неудобно работать), но он есть

Ну и, надеюсь, мне простят подключение светодиодов без балластного резистора? ;)

Файлы к статье

- **Исходники** [2]
- **Файлы для Proteus** [3]

Работа с портами ввода-вывода микроконтроллеров на Си++

При разработке программ для микроконтроллеров (МК) работа с внутренний и внешней периферией является очень важной частью (а иногда и единственной) программы. Это своего рода фундамент, на котором основывается более высокоуровневая логика программы. От эффективности взаимодействия с периферией напрямую зависит эффективность программы в целом. Под эффективностью здесь следует понимать не только скорость выполнения и минимальный размер кода, но и эффективность написания и сопровождения кода.
Многие внешние устройства подключаются к МК через порты ввода-вывода общего назначения (GPIO). Эффективность взаимодействия с этими устройствами во многом зависит от способа работы с портами ввода-вывода.

Тут возникают два, на первый взгляд, противоречивых требования:

- 1)Драйвера внешней периферии хочется писать максимально абстрагировавшись от конкретного способа подключения к микроконтроллеру, а ещё лучше независимо от типа микроконтроллера. Переписывать «библиотечный» код для каждого проекта не очень хорошо.
- 2)Скорость и размер кода в большинстве случаев имеют большое значение.

Ситуация осложняется огромным количеством способов подключения одного внешнего устройства к одному контроллеру и в большинстве случаев определяется удобством изготовления печатной платы наличием свободных линий в порту, необходимостью использовать некоторые линии портов для их специальных возможностей и т.д. Устройство может быть подключено как к одному порту, так и к нескольким, линии порта могут быть как упорядочены по их логическому весу в подключенном устройстве, так и нет. Оно вообще может быть подключено через какой-нибудь расширитель ввода-вывода, например, сдвиговый регистр. При таком разнообразии очень сложно, чтобы драйвер устройства не был заточен под конкретный способ подключения этого устройства, или-же не был слишком громоздким и медленным в случае более универсального решения.

Кажется, что найти универсальное и эффективное решение здесь невозможно. Очень часто в пользу скорости и компактности кода жертвуют и универсальностью и логической структурой программы и как следствие читаемостью кода, просто не отделяя способ подключения устройства от логики его работы. При этом код драйвера модифицируется и подгоняется вручную под каждый проект, под каждый способ подключения.

Давайте разберемся, какие методы работы с портами ввода-вывода традиционно применяются при программировании на чистом Си.
Можно выделить следующие подходы (вариант с жестко заданными в коде именами портов и номерами ножек не рассматриваем):

- 1) Определение портов и линий ввода-вывода с помощью препроцессора.
- 2) Передача порта в код, который его использует, посредствам указателя.
- 3) Виртуальные порты.

Примеры приведены для МК семейства AVR. Компилятор avr-gcc, но описываемые подходы могут быть применены к любым другим МК, для которых имеется стандартный Си/Си++ компилятор.

1. Препроцессор
Способов использования препроцессора для работы с портами в МК существует великое множество. В самом простом и самом распространенном случае просто объявляем порт и номера ножек, к которым подключено наше устройство с помощью директивы #define, не забыв, конечно, про DDR и PIN регистры, если они нужны.
Нет ничего проще, чем помигать светодиодом, подключенным к одному из выводов МК:

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3  #define LED_PORT PORTA
4  #define LED_PIN 5
5
6  int main()
7  {
8          while(1)
9          {
10                 LED_PORT |= 1 << LED_PIN; //зажечь
11                 _delay_ms(100);
12                 LED_PORT &= ~(1 << LED_PIN);
13                 _delay_ms(100);
14         }
15 }
```

Строчка

```
1 LED_PORT |= 1 << LED_PIN;
```

после компиляции превращается в одну команду процессора:

```
1 sbi PORTA, 5
```

также как и

```
1 LED_PORT &= ~(1 << LED_PIN);
```

компилируется в:

```
1 cbi PORTA, 5
```

Выглядит всё очень просто и эффективно. А что, если нам надо управлять несколькими линиями сразу?
Вот пример из хорошо известной библиотеки для работы с дисплеем HD44780 Scienceprog.com Lcd Lib:

```
   #define LCD_RS  0       //define MCU pin connected to LCD RS
1  #define LCD_RW  1       //define MCU pin connected to LCD R/W
2  #define LCD_E   2       //define MCU pin connected to LCD E
3  #define LCD_D4  4       //define MCU pin connected to LCD D3
4  #define LCD_D5  5       //define MCU pin connected to LCD D4
5  #define LCD_D6  6       //define MCU pin connected to LCD D5
6  #define LCD_D7  7       //define MCU pin connected to LCD D6
7  #define LDP PORTD       //define MCU port connected to LCD data pins
8  #define LCP PORTD       //define MCU port connected to LCD control pins
9  #define LDDR DDRD       //define MCU direction register for port connected to LCD data
10 pins
11 #define LCDR DDRD       //define MCU direction register for port connected to LCD control
   pins
```

А вот так там выглядит функция инициализации дисплея:

```
1  void LCDinit(void)//Initializes LCD
2  {
3          _delay_ms(15);
4          LDP=0x00;
5          LCP=0x00;
6          LDDR|=1<<LCD_D7|1<<LCD_D6|1<<LCD_D5|1<<LCD_D4;
7          LCDR|=1<<LCD_E|1<<LCD_RW|1<<LCD_RS;
8
9      //---------one------
10         LDP=0<<LCD_D7|0<<LCD_D6|1<<LCD_D5|1<<LCD_D4; //4 bit mode
11         LCP|=1<<LCD_E|0<<LCD_RW|0<<LCD_RS;
12         _delay_ms(1);
13         LCP&=~(1<<LCD_E);
14         _delay_ms(1);
15
16         //-----------two-----------
17         LDP=0<<LCD_D7|0<<LCD_D6|1<<LCD_D5|1<<LCD_D4; //4 bit mode
18         LCP|=1<<LCD_E|0<<LCD_RW|0<<LCD_RS;
19         _delay_ms(1);
20         LCP&=~(1<<LCD_E);
21         _delay_ms(1);
22
23         //-------three-------------
24         LDP=0<<LCD_D7|0<<LCD_D6|1<<LCD_D5|0<<LCD_D4; //4 bit mode
25         LCP|=1<<LCD_E|0<<LCD_RW|0<<LCD_RS;
26         _delay_ms(1);
27         LCP&=~(1<<LCD_E);
28         _delay_ms(1);
29
30         //--------4 bit--dual line---------------
31         LCDsendCommand(0b00101000);
32     //-----increment address, cursor shift------
33         LCDsendCommand(0b00001110);
34 }
```

Здесь автор ещё пытается записывать биты в порт согласно тому, как они заданы define-ами.
А в функции посылки команды в дисплей автор уже забыл про свои дефайны и молчаливо полагает, что шина данных дисплея подключена строго к старшим четырём разрядам порта:

```
1  void LCDsendCommand(uint8_t cmd)       //Sends Command to LCD
2  {
3          LDP=(cmd&0b11110000);
4          LCP|=1<<LCD_E;
5          _delay_ms(1);
6          LCP&=~(1<<LCD_E);
7          _delay_ms(1);
8          LDP=((cmd&0b00001111)<<4);
9          LCP|=1<<LCD_E;
10         _delay_ms(1);
11         LCP&=~(1<<LCD_E);
12         _delay_ms(1);
13 }
```

В этом случае правильнее было-бы применить побитовый вывод в порт:

```
1  void LCDwrite4(uint8_t cmd)
2  {
3          LDP &= ~(1<<LCD_D7|1<<LCD_D6|1<<LCD_D5|1<<LCD_D4); //clear data bus
4
5          if(cmd & (1 << 0))
6                  LDP |= LCD_D4;
7          if(cmd & (1 << 1))
8                  LDP |= LCD_D5;
9          if(cmd & (1 << 2))
10                 LDP |= LCD_D6;
11         if(cmd & (1 << 3))
12                 LDP |= LCD_D7;
13 }
14
15 void LCDsendCommand(uint8_t cmd)       //Sends Command to LCD
16 {
17         LCDwrite4(cmd);
18         LCP|=1<<LCD_E;
19         _delay_ms(1);
20         LCP&=~(1<<LCD_E);
21         _delay_ms(1);
22         LCDwrite4(cmd);
23         LCP|=1<<LCD_E;
24         _delay_ms(1);
25         LCP&=~(1<<LCD_E);
26         _delay_ms(1);
27 }
```

Так уже будет работать при любом распределении линий шины данных в порту МК, однако размер кода несколько увеличится. С этим уже можно как-то жить. А если для каждой линии завести свой дефайн для имени порта, то таким образом уже можно будет распределить их по разным портам. Размер кода при этом ещё больше раздуется, ведь совмещать записи, даже констант, уже не получится.
Развивая тему с препроцессором можно задавать номер ножки и ее порт в одном определении, ведь Си-шный препроцессор работает не с идентификаторами и не какими-то ни-было языковыми конструкциями, а просто со строковыми литералами.

```
1 #define LCD_RS  PORTA, 0        //define MCU pin connected to LCD RS
2 #define LCD_RW  PORTB, 1        //define MCU pin connected to LCD R/W
3 #define LCD_E   PORTB, 2        //define MCU pin connected to LCD E
```

Добавим к этому средства для манипуляции линией и получим так называемые макросы Аскольда Волкова:

```
1 #define _setL(port,bit)          do { port &= ~(1 << bit); } while(0)
2 #define _setH(port,bit)          do { port |= (1 << bit); } while(0)
3 #define _clrL(port,bit)          do { port |= (1 << bit); } while(0)
```

```
4 #define _clrH(port,bit)          do { port &= ~(1 << bit); } while(0)
5 #define _bitL(port,bit)          (!(port & (1 << bit)))
6 #define _bitH(port,bit)          (port & (1 << bit))
7 #define _cpl(port,bit,val)       do {port ^= (1 << bit); } while(0)
```

Этот подход, в отличии от предыдущих, уже можно сделать переносимым на разные аппаратные платформы. Достаточно только определить эти макросы для целевой платформы соответствующим образом. Однако, у нас по-прежнему остались нерешенными некоторые проблемы. Во-первых, большой размер и низкое быстродействие кода, ведь мы используем побитовый вывод в порты. Даже если часть линий находятся в одном порту и идут подряд, определить эту ситуацию с помощью макросов невозможно. И если, например, размер кода окажется слишком большим, то уже написанную и отлаженную библиотеку придется подгонять под конкретный проект, плодя многочисленные ее версии и, возможно, внося ошибки.
Во-вторых, подключение нескольких однотипных устройств возможно только путём дублирования кода. Опять-же результирующий размер программы неоправданно увеличивается. А потом в две версии одного и того-же кода вносятся изменения независимо друг от друга и каждая из них начинает жить своей жизнью. Спасает здесь только относительно малый размер программ для встроенных систем, ведь если что не так, то можно всё быстренько переписать заново :)

Многие компиляторы для AVR поддерживают побитовый доступ к портам на уровне специальных расширений компилятора (CVAVR, MicroC AVR) или встроенных библиотек (IAR C/C++ Compiler for AVR). Такой побитовый доступ несложно реализовать и в avr-gcc с помощью битовых полей (собственно в IAR примерно так и это и реализовано):

```
1  typedef struct Bits_t
2  {
3          uint8_t Bit0 :1;
4          uint8_t Bit1 :1;
5          uint8_t Bit2 :1;
6          uint8_t Bit3 :1;
7          uint8_t Bit4 :1;
8          uint8_t Bit5 :1;
9          uint8_t Bit6 :1;
10         uint8_t Bit7 :1;
11 }Bits;
12
13 #define PortaBits (*((volatile Bits*)&PORTA))
14 #define LedPin PortaBits.Bit5
15
16 int main()
17 {
18         DDRA = 1 << 5;
19         while(1)
20         {
21                 LedPin = 1;//зажечь
22                 _delay_ms(100);
23                 LedPin = 0;//выключить
24 _delay_ms(100);
25         }
26 }
```

Получается немного чище и удобнее, чем в случае использования макросов, но принципиальных отличий нет. Те-же достоинства и недостатки.

В итоге при использовании препроцессора для манипуляций с портами ввода-вывода мы получаем:

- Простоту и ясность для простых вещей – очень просто написать пару макросов, чтоб поморгать светодиодом.
- Высокую скорость и компактность кода при отказе от универсальности (все ножки в одном порту и желательно по порядку).
- Не расходуется дополнительная память.
- Содержащий большое количество битовых операций код достаточно сложно читать.

- Можно сделать универсально и относительно переносимо пожертвовав размером и скоростью кода (побитовый вывод).
- Для управления несколькими однотипными устройствами придется дублировать код.

2. Передача порта через указатель.
Как уже выяснили один из основных недостатков пи работе с портами ввода-вывода с помощью препроцессора это сложность использования однотипных устройств. Такой сложности не возникает если передавать порт в код, который его использует через указатель.
Порты ввода-вывода в большинстве МК есть не что иное как просто ячейка памяти или регистр в пространстве ввода-вывода и естественно к нему можно обратиться по адресу через указатель.
Удобно запаковать указатель на порт и битовые маски нужных ножек в одну структуру, чтоб потом ее передавать в функцию, которая что-то с ними будет делать. Здесь лучше использовать именно битовые маски, а не битовые позиции, иначе сдвиги вида (1 << some_bit_position) не могут быть вычислены на этапе компиляции (потому, что some_bit_position не константа, а переменная) и будут честно выполнится в каждом месте где встретятся.
Возьмём сдвиговый регистр-защёлку, например 74HC595, который часто используется для экономии выводов МК при подключении многовыводной периферии.

```
1 //ShiftReg.h
2 typedef struct ShiftReg_t
3 {
4         volatile uint8_t *port;
5         uint8_t data_pin_bm;
6         uint8_t clk_pin_bm;
7         uint8_t latch_pin_bm;
8 }ShiftReg;
```

…

```
1  //ShiftReg.c
2  void WriteShiftReg(ShiftReg *reg, uint8_t value)
3  {
4          for(uint8_t i=0; i<8; i++)
5          {
6                  if(value & 1)   //выводим данные
7                          *reg->port |= reg->data_pin_bm;
8                  else
9                          *reg->port &= ~reg->data_pin_bm;
10                 //тактовый импульс
11                 *reg->port |= reg->clk_pin_bm;
12                 value >>= 1;
13                 *reg->port &= ~reg->clk_pin_bm;
14         }
15         //защёлкиваем данные в регистр
16         *reg->port |= reg->latch_pin_bm;
17         *reg->port &= ~reg->latch_pin_bm;
18 }
```

…

```
1  //main.c
2
3  #include <avr/io.h>
4  //вывода data и clk могут быть общие.
5  ShiftReg reg1 = {&PORTA, 1<<1, 1<<2, 1<<3};
6  ShiftReg reg2 = {&PORTA, 1<<1, 1<<2, 1<<4};
7
8  int main()
9  {
10         DDRA = 0xff;
11         DDRB = 0xff;
12         WriteShiftReg(&reg1, 0xff);
13         WriteShiftReg(&reg2, 0x55);
```

```
14          while(1)
15          {
16
17          }
18 }
```

От дублирования кода мы избавились, одна функция WriteShiftReg используется для записи во много сдвиговых регистров. Читаемость кода не пострадала. К тому-же появилась возможность менять порт и ножки к которым подключен регистр во время выполнения программы. Полезность такой возможности, правда, сомнительна особенно для маленьких МК. Таким способом удобно работать с перифирией требующей немного линий ввода-вывода и подключенноу к МК во множественном числе, в том числе подключенные с использованием каких-либо последовательных протоколов USART, SPI (если не хватает аппаратных) 1-Wire и т.д.

Подключение таким образом устройств требующих много линий ввода-вывода непрактично. Код получится слишком громоздким, медленным и требующим много памяти. В данном примере каждый такой сдвиговый регистр занимает 5 байт памяти. Да и побитовый доступ к порту через указатель не самая дешёвая операция:

```
1 *reg->port |= reg->latch_pin_bm;
```

Фрагмент ассемблерного листинга WriteShiftReg

```
1           LD      r30, X+
2           LD      r31, X
3           SBIW    r26, 0x01       ; 1
4           LD      r24, Z
5           ADIW    r26, 0x04       ; 4
6           LD      r25, X
7           SBIW    r26, 0x04       ; 4
8           OR      r24, r25
9           ST      Z, r24
```

Как видно, компилятор не может теперь сопримизировать обращение к порту и установка бита теперь занимает 9 инструкций вместо одной.

Чтобы несколько оптимизировать код можно ввести дополнительные ограничения, например, задать номера ножек константами, и выкинуть соответствующие им битовые маски. В примере со сдвиговым регистром можно заменить константами data_pin_bm и clk_pin_bm и исключить их из структуры, а latch_pin_bm оставить как есть для универсальности:

```
1  typedef struct ShiftReg_t
2  {
3           volatile uint8_t *port;
4           uint8_t latch_pin_bm;
5  }ShiftReg;
6
7  enum {clk_pin_bm = 1 << 0, data_pin_bm = 1 << 1};
8  ...
9  //ShiftReg.c
10 void WriteShiftReg(ShiftReg *reg, uint8_t value)
11 {
12          for(uint8_t i=0; i<8; i++)
13          {
14                  if(value & 1)   //выводим данные
15                          *reg->port |= data_pin_bm;
16                  else
17                          *reg->port &= ~>data_pin_bm;
18                  //тактовый импульс
19                  *reg->port |= clk_pin_bm;
20                  value >>= 1;
21                  *reg->port &= clk_pin_bm;
22          }
```

```
23          //защёлкиваем данные в регистр
24          *reg->port |= reg->latch_pin_bm;
25          *reg->port &= ~reg->latch_pin_bm;
26  }
```

Такая оптимизация сократит код WriteShiftReg примерно на 25 % с незначительной потерей в удобстве.

Итого:

- Удобно использовать для подключения многих однотипных устройств, требующих не много линий ввода-вывода.
- Нет необходимости в дублировании кода.
- Можно менять порт и линии подключения устройства во время выполнения программы.
- Низкая скорость доступа к портам.
- Большой размер кода.
- Требуется дополнительная память для хранения указателя на порт и битовых масок.
- Неудобно и неэффективно работать с большим количеством линий ввода-вывода.

3. Виртуальные порты
Нужно подключить к МК несколько устройств требующих достаточно много линий ввода-вывода, драйвер которых обладает достаточно сложной и объёмной логикой, например, тот-же дисплей HD44780 (при использовании 4х битного интерфейса требует 7 линий). К тому-же устройства могут быть подключены различными способами – к разным линиям портов, или через сдвиговый регистр. Дублировать код драйвера и подгонять его под каждый способ подключения устройства – нет уж, спасибо. Да и размер скомпилированного кода рискует не поместится в целевой МК. Передавать порты драйверу в через указатели? Слишком большие накладные расходы при работе с портами через указатели, много памяти, медленно и громоздко.
Здесь лучше применить, так называемые виртуальные порты. На языке Си они могут быть реализованы как группа функций, принимающих входное значение и выполняющих соответствующие операции ввода-вывода:

```
1  void VPort1Write(uint8_t value)
2  {
3          PORTA = (PORTA & 0xf0) | (value & 0x0f);
4          PORTB = (PORTB & 0x0f) | (value & 0xf0) >> 4;
5  }
6
7  void VPort1DirWrite(uint8_t value)
8  {
9          DDRA = (DDRA & 0xf0) | (value & 0x0f);
10         DDRB = (DDRB & 0x0f) | (value & 0xf0) >> 4;
11 }
12
13 uint8_t VPort1Read()
14 {
15         return (PORTA & 0xf0) | (PORTB & 0x0f) << 4;
16 }
17
18 uint8_t VPort1PinRead()
19 {
20         return (PINA & 0xf0) | (PINB & 0x0f) << 4;
21 }
```

В этом примере входное значение из 8-ми бит распределено между 4-мя младшими битами портов PORTA и PORTB. Реализация функции для вывода команды в дисплей HD44780 с использованием такого виртуального порта будет выглядеть так:

```
1  //объявляем тип указателя на функцию
2  typedef void (*WriteFunc)(uint8_t);
3
4  void LCDwrite4(uint8_t value, WriteFunc write)
5  {
6          enum{LCD_E=4, LCD_RS=5, LCD_RW=6};
```

```
7         uint8_t tmp;
8  tmp = (value & 0x0f) | (1 << LCD_E);//совмещаем вывод тетрады
9         write(tmp);                                    //и установку LCD_E
10        _delay_ms(1);
11        tmp &= ~(1 << LCD_E);
12        write(tmp);
13        _delay_ms(1);
14 }
15
16 void LCDsendCommand(uint8_t cmd, WriteFunc write)     //Sends Command to LCD
17 {
18        LCDwrite4(cmd >> 4, write); //старшая тетрада
19        LCDwrite4(cmd, write);         //младшая тетрада
20 }
```

…

```
1 void DoSomthing()
2 {
3         ...
4         VPort1DirWrite(0xff);
5         LCDsendCommand(0x30, VPort1Write);
6         ...
7 }
```

Теперь становится не важно как именно подключен дисплей к МК. Всё тяжёлая работа по манипуляции портами возложена на функцию VPort1Write. Для каждого способа подключения дисплея нам нужно только написать соответствующую реализацию виртуального порта, тут уже насколько фантазии хватит. Используя, например, уже написанную функцию WriteShiftReg, легко подключить дисплей через сдвиговый регистр:

```
1 ShiftReg reg1 = {&PORTA, 1};
2
3 void VPort2Write(uint8_t value)
4 {
5         WriteShiftReg(&reg1, value);
6 }
```

Такой подход достаточно эффективен для вывода многобитного значения. Из накладных расходов только вызов функции по указателю, но за один вызов выводится сразу много бит. Это эффективнее побитового вывода в порт, но менее эффективно вывода в обычный порт (когда все линии устройства подключены по логическому порядку к одному порту). Тут всё уже зависит от того как реализована функция виртуального порта. Однако если нужно изменить только один бит, то придется запомнить предыдущее значение (или прочитать его из порта), наложить на него соответствующую битовую маску и записать в виртуальный порт. То есть изменения одного бита в этом порту будет дороже записи всех бит порта. Поэтому, например, в функции LCDwrite4 запись тетрады совмещена с установкой бита LCD_E.
Если необходимо не только писать в виртуальный порт, но и читать из него и управлять направлением, подтяжкой или ещё что-то, то функций реализующих виртуальный порт будет много и все их надо писать .

Чего-же мы добились с помощью виртуальных портов:

- Логика работы с устройством полностью отделена от способа подключения устройства.
- Нет необходимости в дублировании кода.
- Сравнительно небольшие накладные расходы на вывод в порт.
- Чистота и понятность кода.
- Необходимо вручную писать реализацию виртуального порта – много однотипных функций (совсем зажрались, подумают некоторые товарищи).
- Манипуляции с отдельными битами неэффективны.

Подход Си++ к работе с портами.
Что может нам предложить язык Си++ при работе с портами ввода-вывода по сравнению чистым Си? Давайте сначала сформулируем, что мы хотим получить в результате наших изысканий:

- 1. Логика работы с устройством должна быть отделена от способа его подключения.
- 2. Не должно быть дублирования кода при подключении многих однотипных устройств.
- 3. Эффективно работать с отдельными битами.
- 4. Эффективно работать с многобитовыми значениями.
- 5. Решение должно быть переносимо на разные аппаратные платформы.
- 6. Не должно использоваться дополнительная память.
- 7. Легкость написания и сопровождения кода.
- 8. Реализация полностью на стандартном Си++.

От динамической конфигурации линий ввода-вывода сразу отказываемся из-за необходимости доступа к портам через указатель со всеми вытекающими последствиями.
Удобно было бы описать линию ввода-вывода в виде отдельной сущности, т.е. класса. В Си++ даже если в классе не объявлено ни одного поля, переменная этого класса всё равно будет иметь размер как минимум один байт, потому, что переменная должна иметь адрес. Значит, нам не надо создавать объекты этого класса, а все функции в нем сделать статическими. А как тогда различать разные линии? Можно сделать этот класс шаблоном, а порт и номер бита передавать в виде параметров шаблона. С номером бита всё ясно – это целое число, его легко передать как нетиповой параметр шаблона. А как быть с портом? Посмотрим, как определены порты ввода-вывода в заголовочных файлах avr-gcc:

```
1 #define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
2 #define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr + __SFR_OFFSET)
3 #define PORTB   _SFR_IO8(0x18)
```

Параметры шаблона могут быть только типовыми или целочисленными константными выражениями, вычисляемыми во время компиляции. Ни указатель, ни ссылку нельзя предать как параметр шаблона:

```
1 template<uint8_t *PORT, uint8_t PIN> //ошибка
2 class Pin
3 {...};
```

Может быть попробовать передавать адрес порта в виде целого числа, а потом его преобразовывать в указатель:

```
1  template<unsigned PORT, uint8_t PIN>
2  class Pin
3  {
4  public:
5          static void Set()
6          {
7                  *(volatile uint8_t*)(PORT + __SFR_OFFSET) |= (1 << PIN);
8          }
9  ...
10 };
```

Это уже работает, но адрес порта придется задавать вручную в виде целого числа, что неудобно:

```
1 typedef Pin<0x18, 1> Pin1;
2 ...
3 Pin1::Set();    //sbi 0x18, 1
```

Взять адрес PORTB по имени не получится потому, что операция взятия адреса не может появляться в константных выражениях, коими должны быть параметры шаблона:

```
1 typedef Pin<(unsigned)&PORTB, 1> Pin1; // ошибка
```

Однако, нам нужно передавать не только адрес порта, ещё нужны PINx и DDRx регистры. К тому-же, в таком виде о переносимости не может быть и речи. Можно, конечно, написать макрос, которому передаём соответствующие имена регистров, и он генерирует соответствующий класс. Но тогда Pin будет слишком жестко завязан на конкретную реализацию портов.
Можно написать перечисление в котором объявлены все порты с удобочитаемыми именами и функцию, которая возвращает нужный порт в зависимости от переданного параметра шаблона.

```
1  enum Ports {Porta, Portb, Portc};
2
3  template<Ports PORT, uint8_t PIN>
4  class Pin
5  {
6  public:
7          static volatile uint8_t & GetPort()
8          {
9                  switch(PORT)
10                 {
11                         case Porta: return PORTA;
12                         case Portb: return PORTB;
13                         case Portc: return PORTC;
14                 }
15         }
16         static volatile uint8_t & GetPin(){...}
17         static volatile uint8_t & GetDDR(){...}
18
19         static void Set()
20         {
21                 GetPort() |= (1 << PIN);
22         }
23 ...
24 };
```

Функцию GetPort можно объявить как внутри класса, так и снаружи. Реализовать её можно, например, с помощью оператора switch или специализаций шаблонной функции:

```
1  template<Ports PORT>
2  volatile uint8_t & GetPort();
3
4  template<>
5  volatile uint8_t & GetPort<Porta>()
6  {
7          return PORTA;
8  }
9
10 template<>
11 volatile uint8_t & GetPort<Portb>()
12 {
13         return PORTB;
14 }
15 ...
16 template<Ports PORT, uint8_t PIN>
17 class Pin
18 {
19 public:
20         static void Set()
21         {
22                 GetPort<PORT>() |= (1 << PIN);
23         }
24 };
```

Однако, такой подход всё равно ограничен. В первую очередь потому, что мы жестко завязываемся на конкретную реализацию портов. Во многих семействах МК для управления портами ввода-вывода имеются дополнительные регистры для быстрого сброса/установки/переключения отдельных бит и много чего ещё. Вот, например, структура, описывающая порт в МК семейства Atmel XMega:

```
1  typedef struct PORT_struct
2  {
3      register8_t DIR;   /* I/O Port Data Direction */
4      register8_t DIRSET;  /* I/O Port Data Direction Set */
```

```
5       register8_t DIRCLR;  /* I/O Port Data Direction Clear */
6       register8_t DIRTGL;  /* I/O Port Data Direction Toggle */
7       register8_t OUT;   /* I/O Port Output */
8       register8_t OUTSET;  /* I/O Port Output Set */
9       register8_t OUTCLR;  /* I/O Port Output Clear */
10      register8_t OUTTGL;  /* I/O Port Output Toggle */
11      register8_t IN;    /* I/O port Input */
12      register8_t INTCTRL;  /* Interrupt Control Register */
13      register8_t INT0MASK;  /* Port Interrupt 0 Mask */
14      register8_t INT1MASK;  /* Port Interrupt 1 Mask */
15      register8_t INTFLAGS;  /* Interrupt Flag Register */
16      register8_t reserved_0x0D;
17      register8_t reserved_0x0E;
18      register8_t reserved_0x0F;
19      register8_t PIN0CTRL;  /* Pin 0 Control Register */
20      register8_t PIN1CTRL;  /* Pin 1 Control Register */
21      register8_t PIN2CTRL;  /* Pin 2 Control Register */
22      register8_t PIN3CTRL;  /* Pin 3 Control Register */
23      register8_t PIN4CTRL;  /* Pin 4 Control Register */
24      register8_t PIN5CTRL;  /* Pin 5 Control Register */
25      register8_t PIN6CTRL;  /* Pin 6 Control Register */
26      register8_t PIN7CTRL;  /* Pin 7 Control Register */
27  } PORT_t;
```

Чтобы изолировать класс Pin от конкретной реализации портов ввода-вывода введём дополнительный уровень абстракции. Добавление нового уровня абстракции вовсе не обязательно влечёт за собой какие-то накладные расходы.
С классом описывающим порт ввода-вывода у нас возникает та-же проблема, что и с классом Pin: как связать класс с конкретными регистрами? Можно конечно попытаться сделать это с помощью перечислений и частичной специализации, но в данном случае это всё-таки лучше сделать с помощью препроцессора:

```
1 #define MAKE_PORT(portName, ddrName, pinName, className, ID) \
2                 class className{\
3                 ...
4                 };
```

Теперь объявим портов на все случаи жизни:

```
1 #ifdef PORTA
2         MAKE_PORT(PORTA, DDRA, PINA, Porta, 'A')
3         #endif
4         ...
5         #ifdef PORT
6         MAKE_PORT(PORTR, DDRR, PINR, Portr, 'R')
7         #endif
```

Проанализировав реализацию портов ввода-вывода различных семейств МК составим минимальный интерфейс для эффективного управления портами (управление режимами подтяжки пока опустим):

```
1  // Псевдоним для типа данных порта.
2  // Для ARM, например, это будет uint32_t.
3  typedef uint8_t DataT;
4
5  // Записать значение в порт PORT = value
6  static void Write(DataT value);
7
8  // Прочитать значение записанное в порт
9  static DataT Read();
10
11 //Записать значение направления линий В/В
12 static void DirWrite(DataT value);
```

```
13
14 // прочитать направление линий B/B
15 static DataT DirRead();
16
17 //Установить биты в порту PORT |= value;
18 static void Set(DataT value);
19
20 // Очистить биты в проту PORT &= ~value;
21 static void Clear(DataT value);
22
23 // Очистить по маске и установить PORT = (PORT & ~clearMask) | value;
24 static void ClearAndSet(DataT clearMask, DataT value);
25
26 // Переключить биты PORT ^= value;
27 static void Togle(DataT value);
28
29 // Установить биты направления
30 static void DirSet(DataT value);
31
32 // Очистить биты направления
33 static void DirClear(DataT value);
34
35 // Переключить биты направления
36 static void DirTogle(DataT value);
37
38 // прочитать состояние линий B/B
39 static DataT PinRead();
40
41 // Уникальный идентификотор порта
42 enum{Id = ID};
43
44 // Разрядность порта (бит)
45 enum{Width=sizeof(DataT)*8};
```

Реализация этого интерфейса для семейств Tiny и Mega AVR будет выглядеть так:

```
1  #define MAKE_PORT(portName, ddrName, pinName, className, ID) \
2                  class className{\
3                  public:\
4                          typedef uint8_t DataT;\
5                  private:\
6                          static volatile DataT &data()\
7                          {\
8                                  return portName;\
9                          }\
10                         static volatile DataT &dir()\
11                         {\
12                                 return ddrName;\
13                         }\
14                         static volatile DataT &pin()\
15                         {\
16                                 return pinName;\
17                         }\
18                 public:\
19                         static void Write(DataT value)\
20                         {\
21                                 data() = value;\
22                         }\
23                         static void ClearAndSet(DataT clearMask, DataT value)\
24                         {\
25                                 data() = (data() & ~clearMask) | value;\
26                         }\
27                         static DataT Read()\
```

```
28                      {\
29                              return data();\
30                      }\
31                      static void DirWrite(DataT value)\
32                      {\
33                              dir() = value;\
34                      }\
35                      static DataT DirRead()\
36                      {\
37                              return dir();\
38                      }\
39                      static void Set(DataT value)\
40                      {\
41                              data() |= value;\
42                      }\
43                      static void Clear(DataT value)\
44                      {\
45                              data() &= ~value;\
46                      }\
47                      static void Togle(DataT value)\
48                      {\
49                              data() ^= value;\
50                      }\
51                      static void DirSet(DataT value)\
52                      {\
53                              dir() |= value;\
54                      }\
55                      static void DirClear(DataT value)\
56                      {\
57                              dir() &= ~value;\
58                      }\
59                      static void DirTogle(DataT value)\
60                      {\
61                              dir() ^= value;\
62                      }\
63                      static DataT PinRead()\
64                      {\
65                              return pin();\
66                      }\
67                      enum{Id = ID};\
68                      enum{Width=sizeof(DataT)*8};\
69              };
```

Поскольку в семействе XMega все регистры порта сгруппированы в одну структуру и есть специальные регистры чтобы быстро устанавливать/очищать/переключать отдельные биты порта, реализация нашего интерфейса будет несколько проще:

```
1  #define MAKE_PORT(portName, className, ID) \
2          class className{\
3          public:\
4                  typedef uint8_t DataT;\
5          public:\
6                  static void Write(DataT value)\
7                  {\
8                          portName.OUT = value;\
9                  }\
10                 static void ClearAndSet(DataT clearMask, DataT value)\
11                 {\
12                         Clear(clearMask);\
13                         Set(value);\
14                 }\
15                 static DataT Read()\
16                 {\
```

```
17                      return portName.OUT;\
18              }\
19              static void DirWrite(DataT value)\
20              {\
21                      portName.DIR = value;\
22              }\
23              static DataT DirRead()\
24              {\
25                      return portName.DIR;\
26              }\
27              static void Set(DataT value)\
28              {\
29                      portName.OUTSET = value;\
30              }\
31              static void Clear(DataT value)\
32              {\
33                      portName.OUTCLR = value;\
34              }\
35              static void Togle(DataT value)\
36              {\
37                      portName.OUTTGL = value;\
38              }\
39              static void DirSet(DataT value)\
40              {\
41                      portName.DIRSET = value;\
42              }\
43              static void DirClear(DataT value)\
44              {\
45                      portName.DIRCLR = value;\
46              }\
47              static DataT PinRead()\
48              {\
49                      return portName.IN;\
50              }\
51              static void DirTogle(DataT value)\
52              {\
53                      portName.DIRTGL = value;\
54              }\
55              enum{Id = ID};\
56              enum{Width=8};\
57      };
58
59 #ifdef PORTA
60 MAKE_PORT(PORTA, Porta, 'A')
61 #endif
62 ...
63 #ifdef PORTR
64 MAKE_PORT(PORTR, Portr, 'R')
65 #endif
```

Анологично можно определить порты B\B для других семейств МК.
Порты B/B теперь инкапсулированы в классы, и мы можем их использовать как типовые параметры шаблонов.
Приступим к реализации класса для линии ввода-вывода:

```
1  template<class PORT, uint8_t PIN>
2          class TPin
3          {
4          public:
5                  typedef PORT Port;
6                  enum{Number = PIN};
7                  static void Set()
8                  {
9                          PORT::Set(1 << PIN);
```

```
10                }
11                static void Set(uint8_t val)
12                {
13                        if(val)
14                                Set();
15                        else Clear();
16                }
17                static void SetDir(uint8_t val)
18                {
19                        if(val)
20                                SetDirWrite();
21                        else SetDirRead();
22                }
23                static void Clear()
24                {
25                        PORT::Clear(1 << PIN);
26                }
27                static void Togle()
28                {
29                        PORT::Togle(1 << PIN);
30                }
31                static void SetDirRead()
32                {
33                        PORT::DirClear(1 << PIN);
34                }
35                static void SetDirWrite()
36                {
37                        PORT::DirSet(1 << PIN);
38                }
39                static uint8_t IsSet()
40                {
41                        return PORT::PinRead() & (uint8_t)(1 << PIN);
42                }
43        };
```

Протестируем полученный класс:

```
1  typedef TPin<Porta, 1> Pa1;
2  ...
3  Pa1::Set();
4  //sbi   0x1b, 1 ; 27
5  Pa1::Clear();
6  //cbi   0x1b, 1 ; 27
7
8  Pa1::Togle();
9  //in    r24, 0x1b       ; 27
10 //ldi   r25, 0x02       ; 2
11 //eor   r24, r25
12 //out   0x1b, r24       ; 27
```

Для удобства определим короткие имена для всех возможных линий B/B:

```
1  #ifdef PORTA
2          typedef TPin<Porta, 0> Pa0;
3          ...
4          typedef TPin<Porta, 7> Pa7;
5          #endif
6          ...
7          #ifdef PORTR
8          typedef TPin<Portr, 0> Pr0;
9          ...
10         typedef TPin<Portr, 7> Pr7;
```

```
11          #endif
```

Как видно, никаких накладных расходов нет, эффективность получилась на уровне того, что можно получит с помощью препроцессора. Те кто давно пишут на Си могут возразить – стоило ли писать какие-то непонятные классы на две страницы вместо того, чтобы написать пару однострочных #define-ов и получить тоже самое?

Конечно-же стоило. Ведь получили мы далеко не тоже самое. Во-первых, класс TPin объединяет в себе все операции применимые к линии В/В. Во- вторых, он жестко типизирован и его можно использовать как параметр шаблона. Например, класс для записи значения в сдвиговый регистр:

```
1  template<class ClockPin, class DataPin, class LatchPin, class T = uint8_t>
2          class ThreePinLatch
3          {
4          public:
5                  typedef T DataT;
6                  enum{Width=sizeof(DataT)*8};
7
8                  static void Write(T value)
9                  {
10                         for(uint8_t i=0; i < Width; ++i)
11                         {
12                                 DataPin::Set(value & 1);
13                                 ClockPin::Set();
14                                 value >>= 1;
15                                 ClockPin::Clear();
16                         }
17                         LatchPin::Set();
18                         LatchPin::Clear();
19                 }
20         };
```

Постой пример его использования:

```
1  typedef ThreePinLatch<Pa0, Pb3, Pc2> Reg1;
2
3  int main()
4  {
5          Pa0::SetDirWrite();
6          Pb3::SetDirWrite();
7          Pc2::SetDirWrite();
8
9          while(1)
10         {
11                 Reg1::Write(PORTD);
12         }
13 }
```

Вызов Reg1::Write компилируется в следующий ассемблерный листинг:

```
1  Reg1::Write:
2          ldi     r25, 0x00
3          sbrs    r24, 0
4          rjmp    .+4
5          sbi     0x18, 3
6          rjmp    .+2
7
8          cbi     0x18, 3
9          sbi     0x1b, 0
10
11         cbi     0x1b, 0
12         subi    r25, 0xFF
```

```
13          cpi     r25, 0x08
14          breq    .+4
15
16          lsr     r24
17          rjmp    .-24
18
19          sbi     0x15, 2
20
21          cbi     0x15, 2
22          ret
```

Сгенерированный листинг не уступает написанному вручную на ассемблере. И кто говорит, что Си++ избыточен при программировании для МК? Попробуйте переписать этот пример на чистом Си, сохранив чистоту и понятность кода, разделение логики работы устройства от конкретной реализации портов В/В и такую-же эффективность.

Списки линий ввода-вывода
Это только начало. Теперь нам предстоит самое интересное — реализовать эффективный вывод многобитных значений. Для этого нам нужна сущность объединяющая группу линий В/В – своеобразный список линий В/В. Поскольку и порты и отдельные линии у нас представлены различными классами, то логично реализовывать список линий с помощью шаблонов. Но здесь есть одна проблема: список линий может содержать различное число линий, а шаблоны в Си++ имеют фиксированное число параметров (а стандарте Cxx03, в следующей версии появятся Variadic templates). Нам поможет библиотека Loki, написанная Андреем Александреску. В ней реализовано множество шаблонных алгоритмов для манипуляций со списками типов произвольной длинны. Это нам подойдёт – списки типов превращаются в списки линий ввода-вывода. Что, собственно, такое списки типов лучше всего почитать у их автора Андрея Александреску в книге Современное проектирование на C++. Очень рекомендую прочитать, хотя-бы мельком, главу «Списки типов» в этой книге. Без этого будет мало понятно, что происходит дальше.

Не во всех МК предусмотрены команду для манипуляций с отдельными битами в портах В/В. В семействе MegaAVR тоже есть порты для которых недоступны битовые операции. Поэтому чтобы сделать операции с портами максимально эффективными нам нужно отказаться от побитового вывода – одно чтение, модификация значения и запись.

То есть нужно записать N битов из входного значения в N битов произвольно расположенных в нескольких портах В/В. Или по другому говоря, сгруппировать записываемые биты по портам и вывести их за раз.

В упрощенном виде алгоритм записи значения в произвольный список линий В/В будет выглядеть так:
1. Определить список портов к которых подключены линии из списка.
2. Для каждого порта:

- Определить список линий к нему подключенный.
- По этому списку сгенерировать битовую маску для битов, которые не нужно менять.
- Спроецировать биты из входного значения на соответствующие им места в регистре порта во временный буфер.
- Наложить битовую маску на регистр порта (т.е. очистить в нем те биту, куда будем записывать новое значение)
- Записать значение из временного буфера в регистр порта.

Выглядит всё это очень сложно. Когда мы пишем реализацию виртуальных портов на Си, то все эти операции проделываем вручную, а сейчас наша задача заставить компилятор выполнять эту работу. Для этого в нашем распоряжении есть списки типов и техника шаблонного метапрограммирования. Пусть компилятор сам тасует биты и считает битовые маски! Приступим.

У каждой линии в списке есть два номера:

- 1. Номер бита во входном значении
- 2. Номер бита в порту, куда он отображается

Оба они понадобятся для того, чтобы спроецировать биты из входного значения в порт.
Второй номер класс TPin помнит сам, оно хранится в enum-е:

```
enum{Number = PIN};
```

Чтобы запомнить первый номер понадобится дополнительный шаблон:

```
template<class TPIN, uint8_t POSITION>
struct PW       //Pin wrapper
{
        typedef TPIN Pin;
        enum{Position = POSITION};
};
```

Хотя можно было этого и не делать, а вычислять битовыю позицию потом с помощью алгоритма IndexOf.

Этот шаблон хранит тип линии B/B и ее битовую позицию в списке. Теперь приступим к генерации собственно списка линий. Для определённости ограничим длину списка 16-ю линиями, ели надо можно добавить и больше, потом. Для этого возьмём шаблон MakeTypelist из библиотеки Loki и модифицируем его под свои нужды:

```
 template
      <
         int Position, // стартовая битовая позиция, сначала это 0
 typename T1  = NullType, typename T2  = NullType, typename T3  = NullType,
 typename T4  = NullType, typename T5  = NullType, typename T6  = NullType,
 typename T7  = NullType, typename T8  = NullType, typename T9  = NullType,
 typename T10 = NullType, typename T11 = NullType, typename T12 = NullType,
 typename T13 = NullType, typename T14 = NullType, typename T15 = NullType,
 typename T16 = NullType, typename T17 = NullType
>
struct MakePinList
{
private:
                // рекурсивно проходим все параметры
                // на следующей итерации Position увеличится на 1,
                // а T2 превратится в T1 и так далее
typedef typename MakePinList
<
                Position + 1,
T2 , T3 , T4 ,
T5 , T6 , T7 ,
T8 , T9 , T10,
T11, T12, T13,
T14, T15, T16,
T17
>::Result TailResult;
                enum{PositionInList = Position};
public:
                // Result это и есть требуемый список линий
typedef Typelist< PW<T1, PositionInList>, TailResult> Result;
};

//конец списка
        //конец рекурсии, когда список пуст
template<int Position>
struct MakePinList<Position>
{
typedef NullType Result;
};
```

В результате на выходе имеем «голый» список типов наших линий ввода вывода. Мы уже можем объявить список из нескольких линий, сделать с ним пока ничего нельзя – для него не определены никакие операции:

```
typedef MakePinList<Pa1, Pa2, Pa3, Pb2, Pb3>::Result MyList;
```

Зато его можно передать в другой шаблон как один параметр. Воспользуемся этим и напишем класс реализующий операции с этим списком:

```
template<class PINS>
struct PinSet
{
...
};
```

Далее напишем алгоритм для преобразования списка линий в список соответствующих им портам:

```
        //шаблон принимает список линий в качестве параметра
 template <class TList> struct GetPorts;

 // для пустого списка результат – пустой тип
 template <> struct GetPorts<NullType>
 {
 typedef NullType Result;
 };

        // для непустого списка
        // конкретизируем, что это должен быть список типов
        // содержащий голову Head и хвост Tail
template <class Head, class Tail>
struct GetPorts< Typelist<Head, Tail> >
{
private:
        // класс TPin помнит свой порт
// запоминаем этот тип порта
                typedef typename Head::Pin::Port Port;
                //рекурсивно генерируем хвост
typedef typename GetPorts<Tail>::Result L1;
public:
        // определяем список портов из текущего порта (Port) и хвоста (L1)
typedef Typelist<Port, L1> Result;
};
```

Теперь мы можем конвертировать список линий в соответствующий список портов, однако один и тот-же порт может содержаться в нем несколько раз. Нам нужны не повторяющиеся порты, по этому воспользуемся алгоритмом NoDuplicates из библиотеки Loki:

```
// конвертируем список линий в соответствующий список портов
typedef typename GetPorts<PINS>::Result PinsToPorts;
// генерируем список портов без дудликатов
typedef typename NoDuplicates<PinsToPorts>::Result Ports;
```

Чтобы организовать рекурсивный проход по списку портов понадобится еще один шаблон класса. Назовём его PortWriteIterator. В качестве шаблонных параметров он принимает список портов и исходный список линий:

```
template <class PortList, class PinList> struct PortWriteIterator;
```

В этом классе и будет находиться реализация операций с отдельными портами. Определим специализацию этого класса для пустого списка линий.

```
template <class PinList> struct PortWriteIterator<NullType, PinList>
{
        // DataType может быть uint8_t или uint16_t (а может и uint32_t в дальнейшем)
        template<class DataType>
        static void Write(DataType value)
        { /*ничего не делаем тут*/ }
```

```
};
```

Далее необходимо выбрать из списка линий, те которые принадлежат определённому порту.

```
// шаблон принимает два параметра:
// TList - список линий
// T - тип порта дл якоторого
template <class TList, class T> struct GetPinsWithPort;

// для пустого списка результат - пустой тип (т.е. тоже пустой список)
template <class T>
struct GetPinsWithPort<NullType, T>
{
typedef NullType Result;
};
// если TList это список типов, голова в котором это PW<TPin<T, N>, M>, т.е. линия в
заданном порту T с битовыми позициями N и M в порту и во входном значении
соответственно, то вставляем её в голову нового списка. Рекурсивно обрабатываем хвост.
template <class T, class Tail, uint8_t N, uint8_t M>
struct GetPinsWithPort<Typelist<PW<TPin<T, N>, M>, Tail>, T>
{
typedef Typelist<PW<TPin<T, N>, M>,
typename GetPinsWithPort<Tail, T>::Result> Result;
};
// если голова списка - любой другой тип, то вставляем на её место рекурсивно
обработанный хвост.
template <class Head, class Tail, class T>
struct GetPinsWithPort<Typelist<Head, Tail>, T>
{
typedef typename GetPinsWithPort<Tail, T>::Result Result;
};
```

Теперь вычислим битовую маску для порта.

```
//Параметр TList должен быть список линий
template <class TList> struct GetPortMask;
// Для пустого списка возвращаем 0.
template <> struct GetPortMask<NullType>
{
enum{value = 0};
};

template <class Head, class Tail>
struct GetPortMask< Typelist<Head, Tail> >
{       //value =       битовая маска для головы  | битовая маска оставшейся части списка
enum{value = (1 << Head::Pin::Number) | GetPortMask<Tail>::value};
};
```

Теперь напишем реализацию для функции записи в порт:

```
// Head - голова списка портов - текущий порт
// Tail - оставшийся список
// PinList - исходный список линий.
template <class Head, class Tail, class PinList>
struct PortWriteIterator< Typelist<Head, Tail>, PinList>
{
//Определим линии принадлежащие текущему порту.
typedef typename GetPinsWithPort<PinList, Head>::Result Pins;
// Посчитаем битовую маску для порта
enum{Mask = GetPortMask<Pins>::value};
typedef Head Port;
```

```
12
13 template<class DataType>
14         static void Write(DataType value)
15 {
16         // проецируем биты из входного значения в соответствующие биты порта
17         // как это реализованно увидим дальше
18                 uint8_t result = PinWriteIterator<Pins>::UppendValue(value);
19                 // если кол-во бит в записываемом значении совпадает с шириной порта,
20                 // то записываем порт целиком.
21                 // это условие вычислится во время компиляции
22                 if((int)Length<Pins>::value == (int)Port::Width)
23                         Port::Write(result);
24                 else
25                 {
26                         // PORT = PORT & Mask | result;
27                         Port::ClearAndSet(Mask, result);
28                 }
29                 // рекурсивно обрабатываем остальные порты в списке
30                 PortWriteIterator<Tail, PinList>::Write(value);
31 }
32 }
```

Функция PinWriteIterator::UppendValue отображает биты во входном значении в соответствующие им биты в порту. Эффективность списков линий в целом зависит в первую очередь именно от реализации этого отображения, поскольку оно, в общем случае, должно выполняться динамически и не всегда может быть вычислено во время компиляции. В зависимости от распределения записываемых бит в результирующем порту применим несколько стратегий отображения бит:

- если записываемые биты в порту расположены последовательно, спроецируем их все сразу с помощью сдвига на нужное число бит и соответствующей битовой маски
- если одиночный бит в исходном значении и в порту имеют одну позицию, спроецируем его с помощью побитного ИЛИ.
- в остальных случаях, если бит во входном значении равен 1, то устанавливаем в 1 соответствующий ему бит в регистре.

```
1  // Tlist – список линий принадлежащих одному порту
2  template <class TList> struct PinWriteIterator;
3  // специализация для пустого списка – возвращаем 0
4  template <> struct PinWriteIterator<NullType>
5  {
6  template<class DataType>
7          static uint8_t UppendValue(const DataType &value)
8          {
9                  return 0;
10         }
11 };
12
13 // специализация для непустого списка
14 template <class Head, class Tail>
15 struct PinWriteIterator< Typelist<Head, Tail> >
16 {
17 template<class DataType>
18 static inline uint8_t UppendValue(const DataType &value)
19 {
20         // проверяем, если линии в порту расположены последовательно
21 // если часть линий в середине списка будет расположена последовательно, то
22 // это условие не выполнется, так, что есть ещё простор для оптимизации.
23         if(IsSerial<Typelist<Head, Tail> >::value)
24         {
25                 // сдвигаем значение на нужное число бит и накладываем не него маску
26                 if((int)Head::Position > (int)Head::Pin::Number)
27                         return (value >> ((int)Head::Position - (int)Head::Pin::Number)) &
```

```
28                                  GetPortMask<Typelist<Head, Tail> >::value;
29                  else
30                          return (value << ((int)Head::Pin::Number - (int)Head::Position)) &
31                                  GetPortMask<Typelist<Head, Tail> >::value;
32          }
33
34          uint8_t result=0;
35
36          if((int)Head::Position == (int)Head::Pin::Number)
37                  result |= value & (1 << Head::Position);
38          else
39                  // это условие будет вычисляться во время выполнения программы
40                  if(value & (1 << Head::Position))
41                          result |= (1 << Head::Pin::Number);
42          // рекурсивно обрабатываем оставшиеси линии в списке
43          return result | PinWriteIterator<Tail>::UppendValue(value);
44  }
45  };
```

Для определения того, что линии в порту расположены последовательно напишем следующий шаблон.

```
1  template <class TList> struct IsSerial;
2  // специализация для пустого списка
3  template <> struct IsSerial<NullType>
4  {
5          // пустой список последователен
6  enum{value = 1};
7  // номер текущей линии
8          enum{PinNumber = -1};
9          // признак конца списка
10         enum{EndOfList = 1};
11 };
12 // для непустого списка
13 template <class Head, class Tail>
14 struct IsSerial< Typelist<Head, Tail> >
15 {
16         // последовательна ли оставшаяся часть списка
17 typedef IsSerial<Tail> I;
18 // запоминаем номер текущей линии в её порту
19 enum{PinNumber = Head::Pin::Number};
20 // не конец списка
21 enum{EndOfList = 0};
22 // список последователен если
23 // номер текущей линии равен номеру следующей - 1 И
24 // оставшаяся часть списка последовательна ИЛИ
25 // текущая линия последняя в списке
26 enum{value = ((PinNumber == I::PinNumber - 1) && I::value) || I::EndOfList};
27 };
```

С учётом всего выше написанного класс PinSet будет выглядеть так:

```
1  template<class PINS>
2  struct PinSet
3  {
4  private:
5          // конвертируем список линий в соответствующий список портов
6          typedef typename GetPorts<PINS>::Result PinsToPorts;
7  public:
8          typedef PINS PinTypeList;
9  // генерируем список портов без дудликатов
10         typedef typename NoDuplicates<PinsToPorts>::Result Ports;
11         // длинна списка линий
```

```
12          enum{Length = Length<PINS>::value};
13          // выбираем тип данных записываемый в список линий
14          // если длинна списка меньше или равна 8 берём тип uint8_t,
15          // если больше - uint16_t
16          typedef typename IoPrivate::SelectSize<Length <= 8>::Result DataType;
17          //записать значение в список линий
18          static void Write(DataType value)
19          {
20                  PortWriteIterator<Ports, PINS>::Write(value);
21          }
22  };
```

Собственно списки линий уже должны работать:

```
1 typedef MakePinList<0, Pa1, Pa2, Pa3, Pb3, Pb4>::Result MyList;
2 typedef PinSet<MyList> MyPins;
3 ...
4 MyPins::Write(0x55);
```

Однако, пользоваться ими пока не очень удобно. Поэтому сделаем вокруг нашей реализации списков линий прозрачную и удобную обёртку:

```
1  template
2  <
3  typename T1  = NullType, typename T2  = NullType, typename T3  = NullType,
4  typename T4  = NullType, typename T5  = NullType, typename T6  = NullType,
5  typename T7  = NullType, typename T8  = NullType, typename T9  = NullType,
6  typename T10 = NullType, typename T11 = NullType, typename T12 = NullType,
7  typename T13 = NullType, typename T14 = NullType, typename T15 = NullType,
8  typename T16 = NullType, typename T17 = NullType
9  >
10 struct PinList: public PinSet
11          <
12                  typename MakePinList
13                  <       0,      T1,
14                          T2 , T3 , T4 ,
15                          T5 , T6 , T7 ,
16                          T8 , T9 , T10,
17                          T11, T12, T13,
18                          T14, T15, T16, T17
19                  >::Result
20          >
21 {
22 // тело этого класса пусое, весь функционал наследован от PinSet
23 };
```

Теперь можно объявлять списки линий следующим образом:

```
1 typedef PinList<Pa1, Pa2, Pa3, Pb3, Pb4> MyPins;
2 MyPins::Write(0x55);
```

Стало достаточно удобно – можно один раз объявить такой список линий, а потом использовать его где угодно, передавать его как шаблонный параметр в классы и функции реализующие управление периферией. А для того, чтобы изменить способ подключения достаточно подправить только одну строку.

Настало время протестировать списки линий с различными конфигурациями и на разных МК.
Выше приведённый пример компилируется в следующий ассемблерный листинг:

```
1  //вывод в PORTA
2          in      r24, 0x1b
```

```
3          andi    r24, 0xF1
4          ori     r24, 0x0A
5          out     0x1b, r24
6  //вывод в PORTB
7          in      r24, 0x18
8          andi    r24, 0xE7
9          ori     r24, 0x10
10         out     0x18, r24
```

Как видно, компилятору все значения были известны и он благополучно посчитал все битовые маски и логические операции, не оставив ничего лишнего на время выполнения. А как он поведёт себя если записываемое значение не известно во время компиляции? Рассмотрим следующий пример (список линий тот-же самый):

```
1  // MCU AtMega16
2  MyPins::Write(PORTC);
3
4  // читаем PORTC
5          in      r18, 0x15       ; 21
6
7  //вывод в PORTA
8          in      r25, 0x1b       ; 27
9          mov     r24, r18
10         add     r24, r24
11         andi    r24, 0x0E       ; 14
12         andi    r25, 0xF1       ; 241
13         or      r24, r25
14         out     0x1b, r24       ; 27
15
16 //вывод в PORTB
17         in      r24, 0x18       ; 24
18         andi    r18, 0x18       ; 24
19         andi    r24, 0xE7       ; 231
20         or      r18, r24
21         out     0x18, r18       ; 24
```

В качестве значения неизвестного используем PORTC. Результат впечатляет, неправда-ли? Даже на ассемблере сложно написать эффективнее. К тому-же если вручную считать все эти битовые маски, то очень велик риск ошибиться.
Рассмотрим тот-же пример скомпилированный для AtXMega128a1. В этом МК порты B/B находятся в расширенном пространстве ввода-вывода и недоступны для операций in и out. Реализация портов, учитывающая особенности семейства XMega у нас уже написана.

```
1 typedef PinList<Pa1, Pa2, Pa3, Pb3, Pb4> MyPins;
2 MyPins::Write(PORTC.IN);
```

```
1  // читаем PORTC.IN
2
3          lds     r18, 0x0648
4
5          mov     r25, r18
6          add     r25, r25
7          andi    r25, 0x0E       ; 14
8
9  //вывод в PORTA
10         ldi     r30, 0x00       ; 0
11         ldi     r31, 0x06       ; 6
12         ldi     r24, 0x0E       ; 14
13         std     Z+6, r24        ; 0x06
14         std     Z+5, r25        ; 0x05
15         andi    r18, 0x18       ; 24
16
17 //вывод в PORTB
18         ldi     r30, 0x20       ; 32
```

```
19          ldi     r31, 0x06       ; 6
20          ldi     r24, 0x18       ; 24
21          std     Z+6, r24        ; 0x06
22          std     Z+5, r18        ; 0x05
```

Как видно, говорить об избыточности и накладных расходах, которые может повлечь использование Си++ здесь не приходится. Аналогичным образом реализуем запись направления линий. Единственное отличие это то, что теперь значение записывается в регистры управления направлением линий (DDRx).

```
1  template<class PINS>
2  struct PinSet
3  {
4          ...
5          static void Write(DataType value)
6          {
7                  PortWriteIterator<Ports, PINS>::Write(value);
8          }
9          //запись направления
10 static void DirWrite(DataType value)
11         {
12                 PortWriteIterator<Ports, PINS>::DirWrite(value);
13         }
14 };
```

И соответствующие дополнения в класс PortWriteIterator.

```
1  template <class Head, class Tail, class PinList>
2  struct PortWriteIterator< Typelist<Head, Tail>, PinList>
3  {
4          ...
5
6  template<class DataType>
7  static void DirWrite(DataType value)
8  {
9                  uint8_t result = PinWriteIterator<Pins>::UppendValue(value);
10                 if((int)Length<Pins>::value == (int)Port::Width)
11                 Port::DirWrite(result);
12                 else
13                 {
14                         Port::DirClear(Mask);
15                         Port::DirSet(result);
16                 }
17                         PortWriteIterator<Tail, PinList>::DirWrite(value);
18         }
19 };
```

Для полноты добавим, аналогичным образом операции Set, Clear, DirSet, DirClear для установки и обнуления битов соответственно в регистрах порта и регистрах направления.

Остаётся реализовать чтение состояния порта. Это несколько проще, чем запись:

- Читаем значение из каждого порта,
- Проецируем биты из порта в выходное значение (в обратном порядке, чем при записи),
- Объединяем значения со всех портов в одно выходное значение.

Подробно на чтении останавливаться не будем.

Итак у нас уже есть средство для эффективной манипуляции портами ввода-вывода МК. Но ведь задача была изначально шире – полностью изолировать драйвера от с способа подключения устройств. И эта цель уже достигнута. Класс TPin, реализующий линию ввода-вывода, как и списки линий полностью изолированы от реализации портов В/В. Это было видно на том примере, что у нас уже реализованы порты как для семейств

Tiny/Mega AVR, так и для XMega. Так вот, достаточно написать класс, реализующий интерфейс порта B/B, и списки линий будут с ним прекрасно работать. Для примера возьмём, уже рассмотренный ранее, сдвиговый регистр и будем его использовать для расширения количества доступных линий B/B.

```
1  template<class ClockPin, class DataPin, class LatchPin, unsigned ID, class T = uint8_t>
2  class ThreePinLatch
3  {
4          public:
5
6          typedef T DataT;
7          // нужен для сортировки портов в списках линий
8          enum{Id = ID};
9          //разрядность регистра
10         enum{Width=sizeof(DataT)*8};
11         // запись значения
12         static void Write(T value)
13         {
14                 _currentValue = value;
15                 for(uint8_t i=0; i < Width; ++i)
16                 {
17                         DataPin::Set(value & 1);
18                         ClockPin::Set();
19                         value >>= 1;
20                         ClockPin::Clear();
21                 }
22                 LatchPin::Set();
23                 LatchPin::Clear();
24         }
25         static DataT Read()
26         {
27                 return _currentValue;
28         }
29         static void ClearAndSet(DataT clearMask, DataT value)
30         {
31                 Write(_currentValue = (_currentValue & ~clearMask) | value);
32         }
33         static void Set(DataT value)
34         {
35                 Write(_currentValue |= value);
36         }
37         static void Clear(DataT value)
38         {
39                 Write(_currentValue &= ~value);
40         }
41         static void Togle(DataT value)
42         {
43                 Write(_currentValue ^= value);
44         }
45         static void DirWrite(DataT value)
46         { /*не можем менять направление*/ }
47         static DataT DirRead()
48         {
49                 return 0xff; //всегда выход
50         }
51         static void DirSet(DataT value)
52         {/* не можем менять направление */}
53         static void DirClear(DataT value)
54         {/* не можем менять направление */     }
55         static void DirTogle(DataT value)
56         {/* не можем менять направление */}
57         protected:
58                 // текущее значение в регистре
59                 static DataT _currentValue;
```

```
60          };
61
62          template<class ClockPin, class DataPin, class LatchPin, unsigned ID, class T>
63          T ThreePinLatch<ClockPin, DataPin, LatchPin, ID, T>::_currentValue = 0;
```

Мы не можем прочитать состояние выходных линий регистра – он всегда работает на выход, поэтому функцию чтения состояния не реализуем и не объявляем. Попытка прочитать состояние такого пора вызовет ошибку компиляции. Зато на запись нет никаких ограничения. И мы свободно можем использовать этот «порт» со списками линий.

```
1  typedef ThreePinLatch<Pa0, Pa1, Pa3, 'R1'> ShiftReg1;
2
3  typedef TPin<ShiftReg1, 0> Rg0;
4  typedef TPin<ShiftReg1, 1> Rg1;
5  typedef TPin<ShiftReg1, 2> Rg2;
6  typedef TPin<ShiftReg1, 3> Rg3;
7  typedef TPin<ShiftReg1, 4> Rg4;
8  typedef TPin<ShiftReg1, 5> Rg5;
9  typedef TPin<ShiftReg1, 6> Rg6;
10 typedef TPin<ShiftReg1, 7> Rg7;
11
12 typedef PinList<Rg4, Rg5, Rg6, Rg7, Pb0, Pb1, Pb2, Pb3> MyPins;
13 ...
14 MyPins::Write(PORTC);
```

Причем нет никаких ограничений на состав списка линий – можно смешивать в одном списке линии сдвигового регистра и линии обычных портов.

В наших целях значится не только работа с многобитными значениями, но и манипуляции отдельными битами, поэтому добавим в интерфейс списков линий функциональность для этого. Во первых, это доступ к отдельным линиям из списка. Для этого в класс PinSet допишем следующий код:

```
1 template<uint8_t PIN>
2 class Pin :public TypeAt<PINS, PIN>::Result::Pin
3 {};
```

Здесь мы воспользовались алгоритмом TypeAt из библиотеки Loki, который возвращает из списка типов указанную в параметре позицию. Пользоваться этим достаточно просто:

```
1 // Установить линию с индексом 1 в единичное состояние.
2 // индексация начинается с 0
3 MyPins::Pin<1>::Set();
4 ...
5 MyPins::Pin<1>::Clear();
```

Обобщая доступ к отдельным битам в списке линий приходим к концепции среза. Не буду вдаваться в подробности их реализации, приведу лишь пример использования:

```
1 // берём срез из MyPins начиная с 4-го бита, длинной 4 бита.
2 typedef MyPins::Slice<4, 4> OtherPins;
```

В результате получаем новый список линий OtherPins, который будет содержать:

```
1 <Pb0, Pb1, Pb2, Pb3>
```

Этот механизм удобен для того, чтобы изменить значения отдельной группы линий, не изменяя остальных. Компилятор для этого случая может сгенерировать более эффективный код. В приведенном примере вызов OtherPins::Write(что-нибудь) запишет значение в 4 бита PORTB, не затронув линии в сдвиговом регистре.

Подведём итоги
Цели поставленные перед реализацией ввода-вывода для МК на Си++ были достигнуты в полной мере. Достигнута как эффективность манипуляций с многобитовыми значениями, так и отдельными линиями ввода-вывода. Списки линий, как и код их использующий, полностью изолированы от конкретной реализации портов. Чтобы добавить новый способ подключения (например через SPI расширитель портов) устройства использующего списки линий, или портировать его на другое семейство МК, достаточно написать соответствующий класс реализующий интерфейс порта ввода-вывода.
Единственными существенными недостатками этого подхода является относительная сложность реализации списков линий (но ведь они уже написаны) и необходимость изучения языка Си++.

Архив с примерами для AVR Studio [1]

Email: klen1@mail.ru Сентябрь 2010

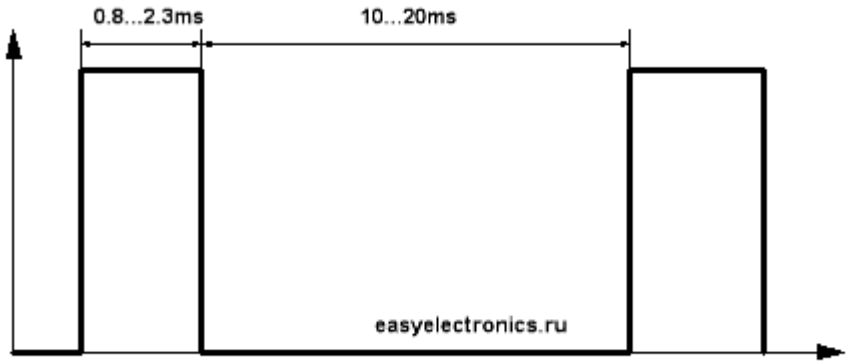
P.S.
Наиболее актуальную версию списков линий (PinList) можно найти по адресу:
github.com/KonstantinChizhov/AvrProjects/tree/master/avrcpp/ [2]

Управление множеством сервомашинок

При построении разных роботов порой приходится использовать несколько сервоприводов. А если это какой-нибудь шестиногий паук, то приводов там этих просто тьма. Как ими управлять? На форуме кое кто даже сокрушался, что ему бы для этих целей плисину применить. Хотя на кой черт там ПЛИСка, когда с рулением даже трех десятков сервоприводов справится самый рядовой микроконтроллер, затребовав под это дело всего один таймер.

Итак, кто не помнит как управляются сервы может прогуляться в старую статью [1] и освежить знания.

Возьмем, для начала, 8 сервомашинок. На каждую серву идет вот такой сигнал:



Управляющий сигнал на сервомашинку

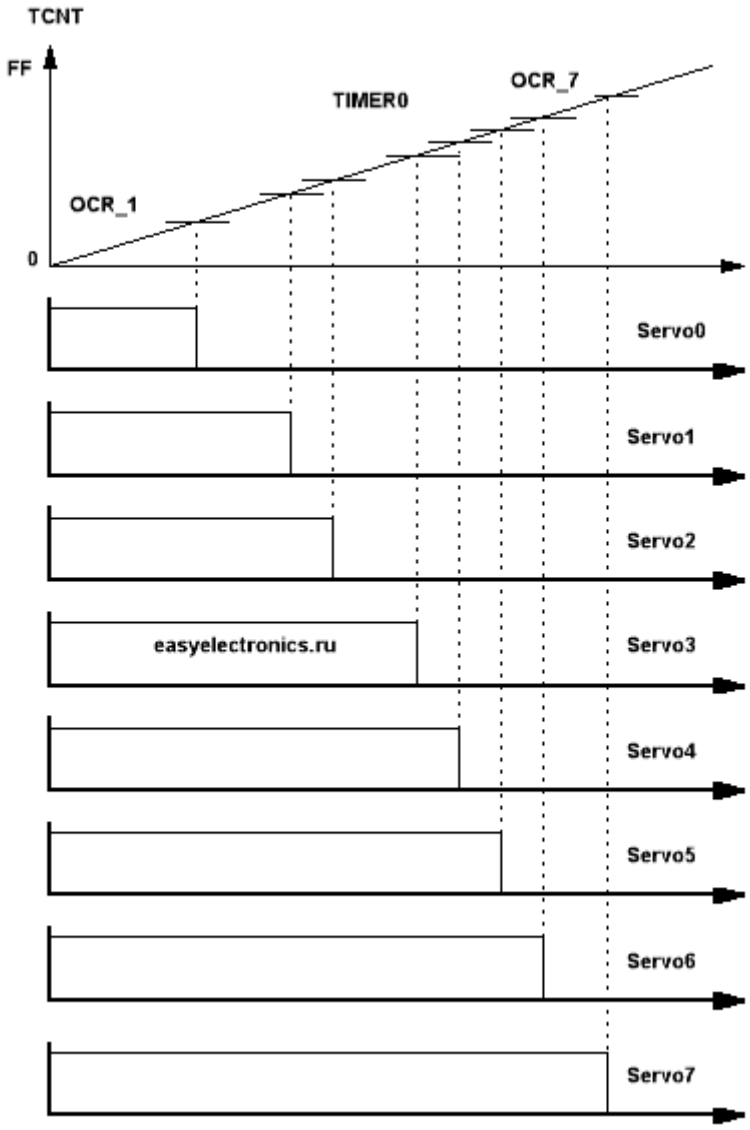
На каждую серву со своей ноги контроллера должна идти такая вот последовательность. Итого подобие ШИМ'а на 8 каналов. Как сгенерировать эту бодягу? Да проще простого. Принцип тут простой. Импульсы медленные — всего то 50Гц, меняются тоже нечасто — серва штука инерционная, поэтому даже сто раз в секунду ей не подергаешь. Так что времени на обработку у нас вагон и маленькая тележка.

Сами импульсы будут генерироваться одним таймером, в фоновом режиме. Принцип генерации прост: Все импульсы стартуют одновременно, выставляя свои уровни в 1.
Затем в таймер, в регистр сравнения, заносится время длительности первого импульса. По прерыванию сравнения происходит:

- Сброс бита на порту первого канала

- Загрузка в регистр сравнения таймера значения длительности второго импульса

И таймер продолжает считать дальше. Загружая по очереди длительности импульсов и отрабатывая все каналы. Когда досчитает до конца, то надо настройкой прескалера замедлить его ход и отщелкать 20мс, а затем повторить цикл.



Таким образом, у нас все восемь каналов формируются в прерывании конечным автоматом, который сам себя загружает, сам сбрасывает биты и так далее. Практически не тратя временные ресурсы контроллера, беря на себя управление только тогда, когда надо сменить уровень на ноге, а это очень малая по времени операция и очень редкое, в масштабах контроллера, событие.

Есть лишь одна деталь — для успешной работы каналы серв должны быть сортированы по возрастанию. Чтобы можно было просто последовательно загружать значение и все.

Конечный автомат, висящий на прерывании сравнения состоит из двух таблиц с данными:

```
1 #define MaxServo 8              // Число сервомашинок
2 u08 servo_state=0;              // Состояние конечного автомата.
3 u08 ServoPortState[MaxServo+1];       // Значение порта которое надо вывести
4 u08 ServoNextOCR[MaxServo+1];  // Время вывода значения
```

ServoPortState — содержит биты которые надо сбросить при каждом временном интервале. То есть если у нас два сервоканала имеют одинаковую длительность импульсов (т.е. сервы выставлены в одно и то же положение), то они будут сброшены одновременно. За один проход.
ServoNextOCR — таблица длительностей сигналов. Тут в порядке возрастания собраны длительности отчетных интервалов.

И код автомата:

```
   ISR(TIMER0_COMP_vect)                          // Прерывание по совпадению
   {
   if (servo_state)                               // Если не нулевое состояние то
           {
1          OCR0 = ServoNextOCR[servo_state];      // В регистр сравнения кладем следующий
2  интервал
3          PORTA &= ~ServoPortState[servo_state]; // Сбрасываем биты в порту, в соответствии
4  с маской в массиве масок.
5          servo_state++;                         // Увеличиваем состояние автомата
6
7          if (OCR0 == 0xFF)                              // Если значение сравнения равно FF
8  значит это заглушка
9                  {                              // И мы достигли конца таблицы. И пора
10 обнулить автомат
11                 servo_state = 0;                       // Выставляем нулевое состояние.
12
13                 TCNT0 = 105;                   // Программируем задержку в 20мс (на
14 предделителе 1024)
15                 TCCR0 &= 0b11111000;           // Сбрасываем предделитель таймера
16                 TCCR0 |= 0x05;                 // Устанавливаем предделитель на 1024
17
18                 if (servo_need_update)         // Если поступил приказ обновить таблицы
19 автомата
20                         {
21                         Servo_upd();           // Обновляем таблицы.
22                         servo_need_update = 0; // Сбрасываем сигнал обновления.
23                         }
24                 }
25         }
26 else                                           // Нулевое состояние автомата. Новый цикл
27         {
28         OCR0 = ServoNextOCR[servo_state];              // Берем первую выдержку.
29
30         TCCR0 &= 0b11111000;                   // Сбрасываем предделитель
31         TCCR0 |= 0x04;                         // Предделитель на 256
32         PORTA = 0xFF;                          // Выставялем все сервоканалы в 1 - начало
33 импульса
34
           servo_state++;                         // Увеличиваем состояние конечного
   автомата.
           }
   }
```

Вот такой вот незамысловатый алгоритм. Это касаемо генерации самих импульсов. Но ведь нам надо еще каким то образом сделать таблицы длительностей и битмасок, которые будут формировать сигнал. Да еще обеспечить человеческий интерфейс взаимодействия с конечным автоматом. Этим и займемся :)

Управление сервомашинками осуществляется через структуру под которую мы даже зарядил свой тип:

```
1 typedef struct {
2                 u08 Position;
3                 u08 Bit;
4                 }  SArray_def;
5
```

```
6 SArray_def Servo[MaxServo];
7 SArray_def *Servo_sorted[MaxServo];
```

Servo.bit — содержит бит порта,который нам надо сбросить. Просто единичка в нужном разряде.
Servo.Position — это позиция сервопривода. В моем случае может принимать значения от 26 до 76. Если больше-меньше, то серва уйдет за границы допустимых положений, что черевато. Можно было поиграться с частотой процессора и предделителями, чтобы выбрать диапазон побольше, но мне было лень. В конце концов, для демонстрации возможностей 50 положений вполне хватит. А желающие могут и на 16ти разрядный таймер все перевесить и будет у них точность до долей градуса :)

Так как сервомашинки надо сортировать, то я сделал еще массив указателей Servo_sorted. Чем ворочать в памяти неповоротливые структуры, я лучше перетасую в порядке возрастания поля Position их указатели. Сожрет лишнюю память, но выйграет вы быстродействии.

Впрочем, в данном случае, когда структура имеет всего два восьмибитных поля, экономия выглядит сомнительной, т.к. тот же указатель занимает те же самые два байта. В лучшем случае баш на баш. Но кто сказал, что структура будет неизменной? Может что еще захотим добавить! Поэтому пусть лучше сортируются указатели.

Для начала указатели и структуры инициализируются нужными данными:

```
1  // Присваиваем указателям адреса наших структур.
2  Servo_sorted[0] = &Servo[0];
3  Servo_sorted[1] = &Servo[1];
4  Servo_sorted[2] = &Servo[2];
5  Servo_sorted[3] = &Servo[3];
6  Servo_sorted[4] = &Servo[4];
7  Servo_sorted[5] = &Servo[5];
8  Servo_sorted[6] = &Servo[6];
9  Servo_sorted[7] = &Servo[7];
10
11 // Заполняем битмаски
12 Servo[0].Bit = 0b00000001;
13 Servo[1].Bit = 0b00000010;
14 Servo[2].Bit = 0b00000100;
15 Servo[3].Bit = 0b00001000;
16 Servo[4].Bit = 0b00010000;
17 Servo[5].Bit = 0b00100000;
18 Servo[6].Bit = 0b01000000;
19 Servo[7].Bit = 0b10000000;
20
21 // Выставляем все положения на нейтраль -- точно посредине диапазона.
22 Servo[0].Position = 50;
23 Servo[1].Position = 50;
24 Servo[2].Position = 50;
25 Servo[3].Position = 50;
26 Servo[4].Position = 50;
27 Servo[5].Position = 50;
28 Servo[6].Position = 50;
29 Servo[7].Position = 50;
```

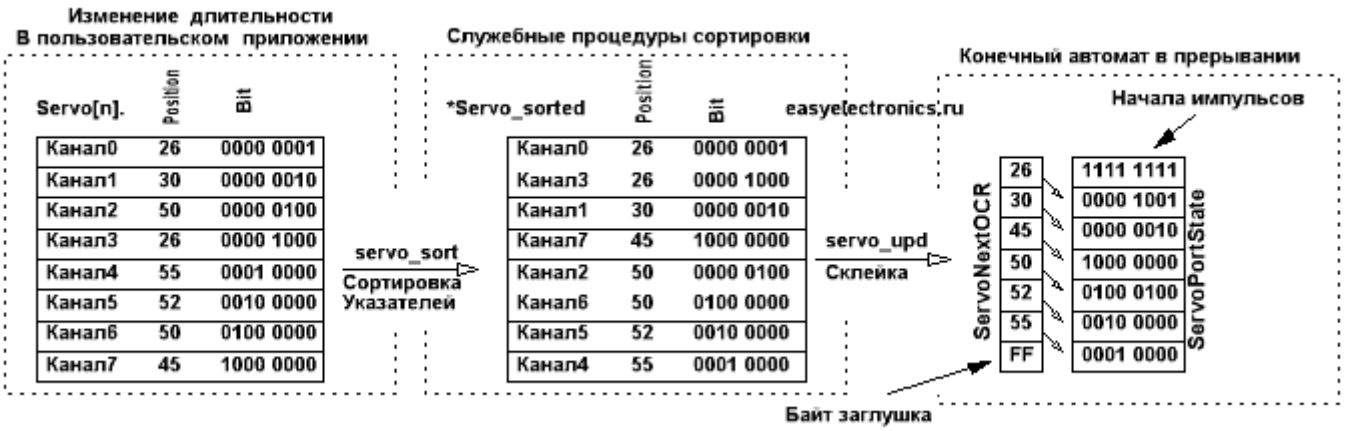
Теперь можно писать из прикладной программы любые произвольные значения (26…76 в данном случае) в поле Position, задавая произвольное положение любой из восьми сервомашинок.

После изменения любого из полей, перед отправкой таблиц в конечный автомат, данные надо упорядочить по длительности. Это делает спец функция.

```
1  //Простейший алгоритм сортировки вставкой. Недалеко ушел от пузырька, но на столь малых
2  количествах
3  // данных являетя наиболее эффективным.
4  void Servo_sort(void)
5  {
```

```
6  u08 i,k;
7  SArray_def *tmp;
8
9  // Сортируем массив указателей.
10 for(i=1;i<MaxServo;i++)
11         {
12         for(k=i;((k>0)&&(Servo_sorted[k]->Position < Servo_sorted[k-1]->Position));k--)
13                 {
14                 tmp = Servo_sorted[k];                                  // Swap [k,k-1]
15                 Servo_sorted[k]=Servo_sorted[k-1];
16                 Servo_sorted[k-1]=tmp;
17                 }
18
19         }
   }
```

Массив отсортирован. Теперь надо его обработать и сформировать таблицы для автомата. Обработка заключается в склейке совпадающих полей. Т.е. если у нас есть, например, 5 каналов с длительностью 20, то их надо склеить в одно значение таблицы, а биты слепить в единую битмаску. А потом загрузить все в массивы конечного автомата.



Обновление идет в функции Servo_upd

```
1  void Servo_upd(void)
2  {
3  u08 i,j,k;
4
5  for(i=0,k=0;i<MaxServo;i++,k++)
6  {
7          if(Servo_sorted[i]->Position!=Servo_sorted[i+1]->Position)    //Если значения
8  уникальные
9          {
10         ServoNextOCR[k] = Servo_sorted[i]->Position;                  // Записываем их как
11 есть
12         ServoPortState[k+1] = Servo_sorted[i]->Bit;                   // И битмаску туда
13 же
14         }
15         else                                                         // Но если совпадает
16 со следующим
17         {
18         ServoNextOCR[k] = Servo_sorted[i]->Position;                  // Позицию
19 записываем
20         ServoPortState[k+1] = Servo_sorted[i]->Bit;                   // Записываем
21 битмаску
```

```
22
23          // И в цикле ищем все аналогичные позиции, склеивая их битмаски в одну.
24
25          for(j=1;(Servo_sorted[i]->Position == Servo_sorted[i+j]-
26 >Position)&&(i+j<MaxServo);j++)
27                  {
                    ServoPortState[k+1] |= Servo_sorted[i+j]->Bit;
                    }
            i+=j-1;                                         // Перед выходом корректируем
   индекс
            }                                               // На глубину зарывания в повторы
   }
   ServoNextOCR[k] = 0xFF;                                          // В последний элемент
   вписываем заглушку FF.
   }
```

Т.к. менять таблицы автомата до тех пор пока он не отработает до конца черевато глюками и ахтунг работой, то делать обновление можно только только тогда, когда автомат выйдет на нулевое положение. Поэтому я повесил это дело в прерывание. А в прикладной программе оставил только обновление структур и их сортировку. И добавил флаг готовности servo_need_update.

Т.е. если нам надо изменить позицию сервомашинки, то мы пишем следующую комбинацию:

```
1 Servo[Б].Position =A
2 Servo_sort();
3 servo_need_update = 1;
```

И при следующем проходе автомата сервомашинка Б выйдет на позицию А. Можно одновременно менять хоть все машинки сразу.

Я сделал небольшую демонстрашку этой программы. Через диспетчер пустил по восьми сервам синусоидальные волны. Если сделать роботиста подобного тому, что уже описывался в моем блоге [2], то наверное он поползет по змеиному. Впрочем, я не уверен в этом. Т.к. сколько я не наблюдал змей никак не могу понять физическое обоснование их движению. Она словно по земле течет.

Сама программа, гонящая синус по сервам, сделана на диспетчере и выглядит так:

```
1 //Sinus
2 u08 SinState=0;                // Состояние генератора
3
4 // Таблица синуса, приведенная к нашим допустимым значениям
5 u08 SinTable[] PROGMEM = {76,74,68,60,50,41,33,27,26,29,35,43,53,62,70,75};
```

```
1                               void Sinus(void)
2                               {
3
4                               SetTimerTask(Sinus,100);                    // Перезапуск
5                               задачи через 100мс
6
7                               Servo[0].Position =
8                               pgm_read_byte(&SinTable[SinState&0x0F]);
9                               Servo[1].Position =
10                              pgm_read_byte(&SinTable[(SinState+1)&0x0F]);
11                              Servo[2].Position =
12                              pgm_read_byte(&SinTable[(SinState+2)&0x0F]);
13                              Servo[3].Position =
14                              pgm_read_byte(&SinTable[(SinState+3)&0x0F]);
15                              Servo[4].Position =
16                              pgm_read_byte(&SinTable[(SinState+4)&0x0F]);
17                              Servo[5].Position =
18                              pgm_read_byte(&SinTable[(SinState+5)&0x0F]);
19                              Servo[6].Position =
```

```
pgm_read_byte(&SinTable[(SinState+6)&0x0F]);
Servo[7].Position =
pgm_read_byte(&SinTable[(SinState+7)&0x0F]);

SinState++;                                          //
Увеличиваем состояние

Servo_sort();                                        // Сортируем
таблицы
servo_need_update = 1;                               // Запрос на
обновление автомата
}
```

Обратите внимание на реализацию генерации синуса. У меня в таблице синуса (вообще то там косинус, но пофигу) есть только один период. Из 16 значений. А у нас восемь серв на которые должны идти последовательные значения из этого ряда:

- n
- n+1
- n+2
- n+3
- n+4
- n+5
- n+6
- n+7

Но т.к. период всего один и вывод надо закольцовывать. Т.е. если n+m вылазит за границу длины нашего масива, то его надо завернуть в начало. Если делать это математическим методом условий, то будет каша с кучей проверок.

Я, как бывалый ассемблерщик, решил все проще, применив трюк с ограничением разрядности битмасками. Т.е. в массиве у нас 16 значений. Поэтому если мы n+m обрежем до 4 бит маской, то оно в принципе не сможет вылезти за границы в 16 (0x0F) элементов и так и будет вращаться по кругу. Даже проверку на **if (n==7) n=0;** делать не надо. Усе будет автоматически!

Данный метод должен прорулить и в кольцевых буферах, кратных 4, 8, 16, 32, 64, 128, 256 байтам. Просто записываем базу, а потом имеем смещения головы и хвоста, ограниченные в разрядности, чтобы они автоматом не вылезали за пределы кольца.

Конструкция pgm_read_byte — это чтение из флеша. Подробно я ее уже расписывал в статье про работу с памятью [3]

Проблемы
Главная проблема при такой генерации — это джиттер. Т.е. дрожание фронтов. Возникает из-за того, что прерывания в AVR имеют одинаковый приоритет и кто раньше встал того и тапки. А если в системе есть еще один таймер, генерящий свое прерывание? То он будет создавать задержки для конечного автомата, из-за чего поплывут фронты. Что черевато. У меня такое прерывание есть — таймер диспетчера задач с его службой времени. Я решил проблему просто — диспетчер рулит прикладными задачами, поэтому его приоритет может быть и ниже автомата, который обеспечивает хардверную реализацию фичи.

Поэтому я разрешил прерывания в обработчике прервыаний диспетчера. Получились вложенные прерывания. В принципе ничего страшного, главное понимать, чем это грозит. А грозит это зарыванием в прерываниями — это когда прерывание вызывается чаще чем обрабатывается. И более глубоким загрузом стека. Зарывание исключено — Служба таймеров обрабатывается в пару порядков быстрей чем вызывается, а вот от двойной нагрузки на стек никуда не деться. Но это можно просто учесть.

Архив с проектом для WinAVR+GCC [4]

Скомпилировали, зашили, запустили…
Получается такая картина на выходах портов:

Дофига серв у меня, к сожалению, нет. Есть всего одна. Зато есть осциллограф с логическим анализатором. Поэтому я его просто подключил к схеме и снял видяшку:

Данную программную конструкцию можно легко расширить хоть до сотни серв. Главное лишь бы портов хватило. Надо всего лишь изменить разрядность данных отвечающих за битмаски, а в конструкции автомата, в части где идет вывод:

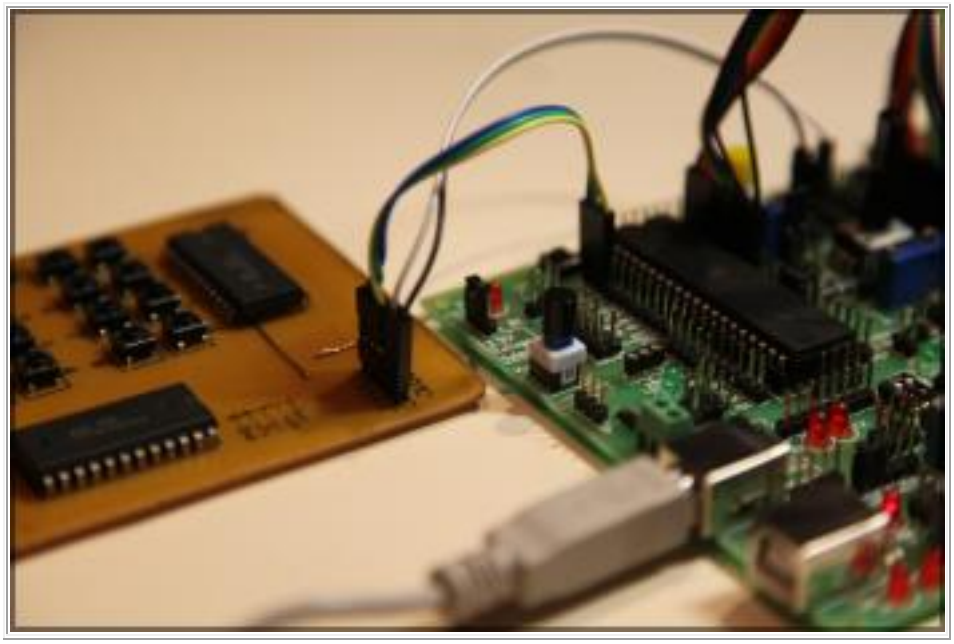
```
1         PORTA &= ~ServoPortState[servo_state];
```

Добавить еще портов и выводить на них соответствующие байты многобайтной битмаски. Можно это сделать маской-сдвигом, но наверное лучше будет через указатель вытаскивать нужные байты.

И при этом у нас остается еще прорва процессорного времени на выполнение других задач!

З.Ы.
Оптимизацией кода практически не занимался, так что там еще многое что следует доточить. Начиная от механизма инверсии битов и заканчивая оптимизацией всех структур.

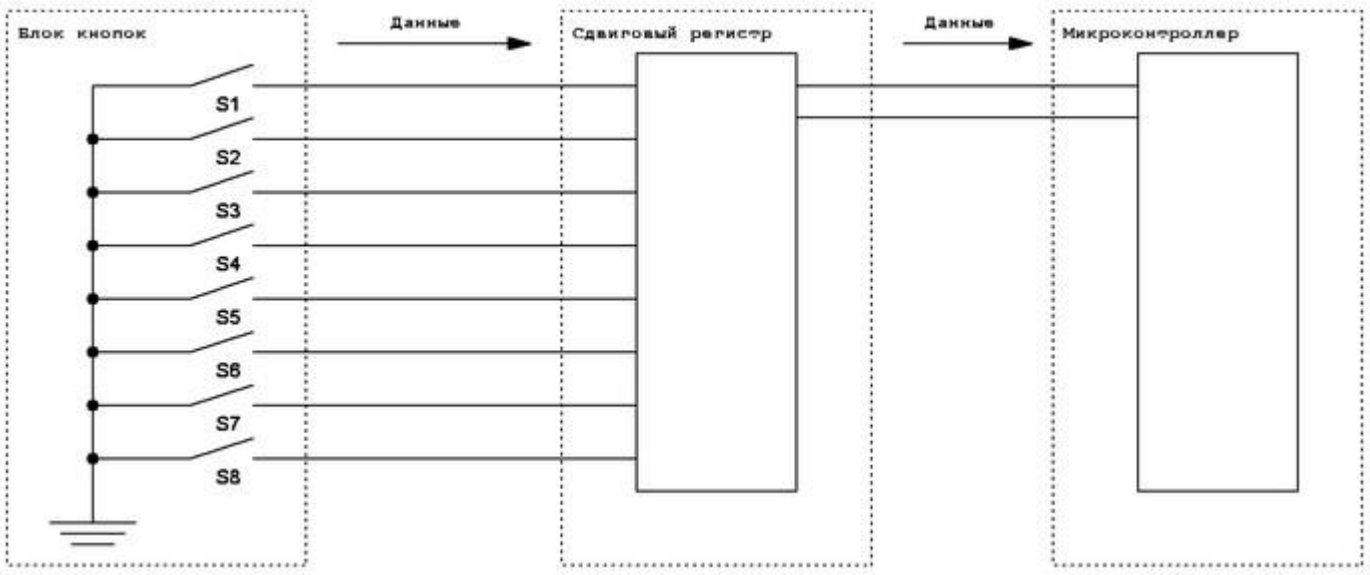
Подключение клавиатуры к МК по трем проводам на сдвиговых регистрах



Часто возникает необходимость использования в своем проекте большого количества кнопок для различный целей. Существуют разные варианты реализации данной задачи. Сегодня я расскажу вам о решении, которое пришло мне в голову однажды вечером. Тогда мне нужно было повесить клавиатуру на контроллер с ограниченным количеством свободных ножек. Скорее всего данное решение уже описывалось и использовалось где-либо ранее, но упоминаний в интернете я не нашел.

Суть
Подключение клавиатуры осуществляется по трем сигнальным проводам. Дополнительные элементы: сдвиговые регистры **sn74198n** и несколько резисторов. Максимальное количество кнопок ограничивается лишь максимально допустимым временем на сканирование клавиатуры. Я использовал в своем проекте 16 кнопок, но путем добавления новых сдвиговых регистров, можно увеличить данное число до необходимого вам значения. Вас будет ограничивать лишь пропорционально возрастающее время сканирования клавиатуры.

Аппаратная часть
Сдвиговые регистры – вещь довольно удобная за счет своей дешевизны и универсальности. Их часто используют для подключения светодиодов, семисегментных индикаторов и т.п. по небольшому количеству выводов микроконтроллера. В нашем случае, будем проделывать почти все тоже самое, но в обратную сторону:будем передавать данные не «из микроконтроллера», а «в него». Для наглядности привожу блок схему работы данного устройства:



- 1.Блок Клавиатуры
- 2.Сдвиговый регистр
- 3.МК

Блок клавиатуры представляет собой набор кнопок, которые одним выводом подключены к земле, а другим подключаются к соответствующему входу сдвигового регистра.

Параллельная и последовательная передача
Если вы не знаете различия в параллельном и последовательном способе передачи информации, то рекомендую разобраться в этом сейчас. Далее я часто буду использовать эти термины, и без понимания их, вам будет немного сложнее.
Параллельной передачей данных называют метод передачи нескольких сигналов с данными одновременно по нескольким параллельным каналам, например порт целиком, сразу все восемь бит. При последовательной передачи данных биты пересылаются по одной линии связи, друг за другом, последовательно. Например по UART или по SPI

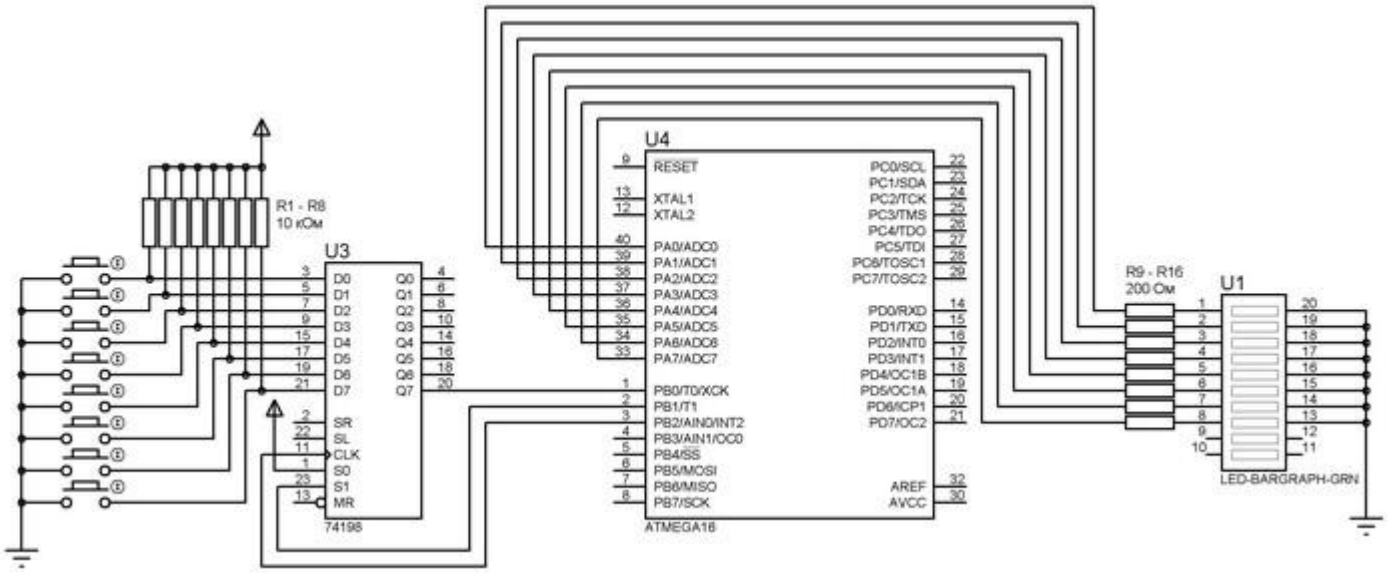
В нашей схеме сдвиговый регистр осуществляет конвертацию параллельного входного сигнала в последовательный выходной. В процессе сканирования клавиатуры он будет хватать 8 значений из блока клавиатуры и последовательно отсылать его в микроконтроллер.
Исходя из вышеописанного, к сдвиговому регистру предъявляются следующие требования:

- параллельный вход
- последовательный выход

Для этого нам подойдет регистр SN74198N. В моем месте обитания такие микросхемы редкость, поэтому я использовал советские монстровидные К155ИР13. Они являются полным аналогом, и различий в их использовании от импортных собратьев нет.

МК управляет сдвиговым регистром: говорит когда ему хватать сигнал с блока клавиатуры и передавать его своему хозяину.

Теперь разберем схему нашего устройства.



Крупным планом [1]

Схема упрощенная, показано только подключение клавиатуры. Питание и прочая обвязка контроллера как обычно.

Описание кнопок
Одним выводом кнопки подключены к входу сдвигового регистра. Другим – к общему проводу. Так же немаловажная детать – подтягивающие резисторы R1 – R8. Они создают высокий логический уровень на входе сдвигового регистра в то время, как соответствующая ему кнопка разомкнута.
Как только кнопка замыкается, на входе сдвигового регистра образуется логический ноль, так как он оказывается напрямую подключен к общему проводу. Значение резистора в 10 кОм не дает протекать слишком большому току, пока кнопка замкнута, и достаточно хорошо создает высокий логический уровень, пока кнопка разомкнута.

Сдвиговый регистр

- Наш сдвиговые регистр имеет входной порт (D0-D7)
- выходной порт (Q0-Q7)
- и сигналы управления (SR, SL, CLK, S0, S1, MR).

Входной и выходной порт можно использовать как в параллельном режиме, так и в последовательном. За счет этого, данный сдвиговый регистр достаточно универсален. Но мы лишь остановимся на тех функциях, которые нам необходимы, остальное вы можете сами прочитать в даташите.

- D0-D7 – собственно вход. Сюда мы будем подавать 8 сигналов с наших кнопок.
- Q0-Q7 – параллельный выход. Мы будем использовать лишь одну ногу – Q7. Остальные в нашем случае нам не нужны.
- CLK – тактовый вход. Все в сдвиговом регистре делается только по дрыгу на этой ножке. А точнее по нарастающему фронту. Если нам необходимо что-то сделать, то просто подаем высокий уровень на ножку CLK, а затем опускаем ее (подаем логический ноль). Для простоты изложения, я буду по деревенский говорить, что нам нужно кликнуть ножкой CLK.
- S0 и S1. Эти ножки отвечают за поведение выхода регистра (Q0-Q7). По сути этот регистр может иметь как последовательный (наш случай), так и параллельный выход. Когда обе ножки подняты (высокий логический уровень), мы можем загрузить на выход те данные, которые в данный момент находятся на входе (D0-D7).

  Для этого нам нужно при поднятых S0 и S1 кликнуть ножкой CLK. Сразу же после этого, ножки на выходе примут те же значения, что и на входе.

  Если же опустить одну ножку в ноль, например S1, то стоит нам дернуть ножкой CLK, как данные на выходном порте сдвинутся в сторону старшего бита (бит с выхода Q0 перескочит на выход Q1, а бит, который раньше был на месте Q1 — перелезет на Q2 и т.д.). Кликнем еще раз, биты сдвинутся опять.

Проделав это 8 раз, наши данные убегут вникуда, а на смену им прийдут новые, взявшиеся ниоткуда (откуда они берутся, мы поговорим позже). Дабы не терять эти данные, а передавать их в микроконтроллер, нужно вовремя их считывать.

Так как наши биты двигаются в сторону старшего (Q7), то и ловить их следует там.

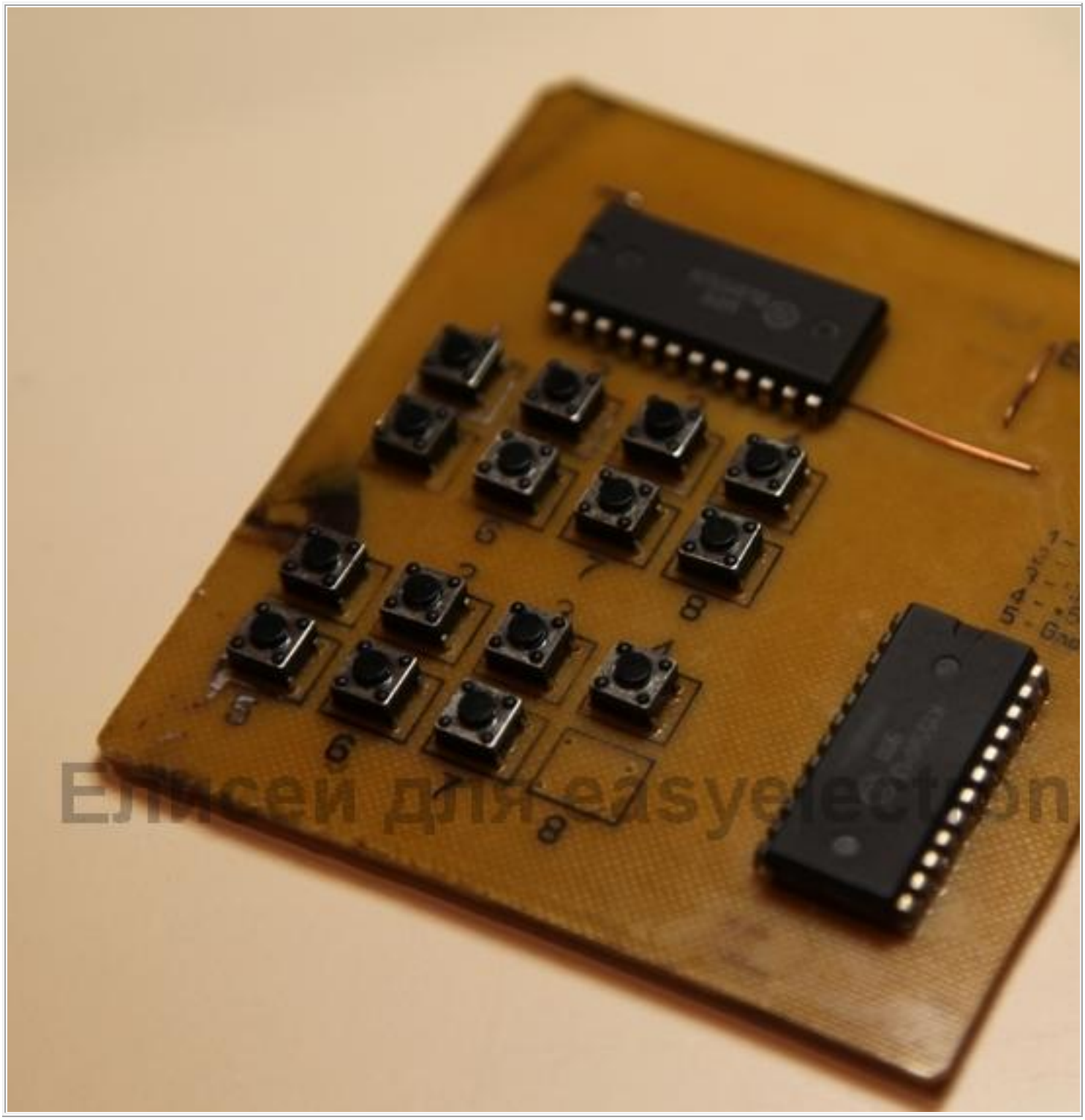
Алгоритм действий

- 1. Ставим S0 и S1 в высокий уровень и кликаем. На выходе Q0-Q7 у нас фиксируется необходимый нам байт (8 бит).
- 2. Опускаем S1 (S0 остается в высоком уровне всегда, поэтому его можно смело напрямую сажать на плюс и забыть про него)
- 3. Считываем бит с ножки Q7
- 4. Кликаем. Бит, который раньше был на ножке Q7, убегает в нибытие, а на смену ему приходит бит, который раньше стоял на ножке Q6.
- 5. Возвращаемся на пункт 3 и проделываем эти операции еще 7 раз. Ибо у нас 8 бит, и нам важен каждый из них.

Итого, использую 3 ножки микроконтроллера, подключенных к сдвиговому регистру (CLK, S1, Q7), мы легко и непринужденно опрашиваем наши 8 кнопок.

На бумаге это все работает, но проверим это в реальном железе. Для этого я собрал демонстрационную платку и подключил ее к Pinboard. Использовал, как я уже говорил, советские аналоги сдвиговых регистров, а именно К155ИР13, поэтому плата разведена под них.

Не пугайся что на плате 16 кнопок и уже 2 сдвиговых регистра, позже я опишу и этот вариант схемы. Главное знай, что данная схема может безболезненно работать и в восьмикнопочном режиме.



16-ая кнопка у меня отсутствует по причине отсутствия ее у меня в запасах и мороза на улице. Плату я перекатал ЛУТом на текстолит, вытравил и запаял. Вся эта технология хорошо расписана на данном сайте, поэтому останавливаться на деталях не буду.

Итак, плата готова, приступаем к самому интересному – написанию прошивки.

Програмная часть
Хотя представленный вариант работы со сдвиговым регистром совместим с аппаратным SPI микроконтроллера, опрашивать кнопки будем программно. Напишем функцию, которая будет опрашивать наши кнопки, и помещать результат в регистр R17.

```
.equ    BTN_PORT        =       PORTB
```

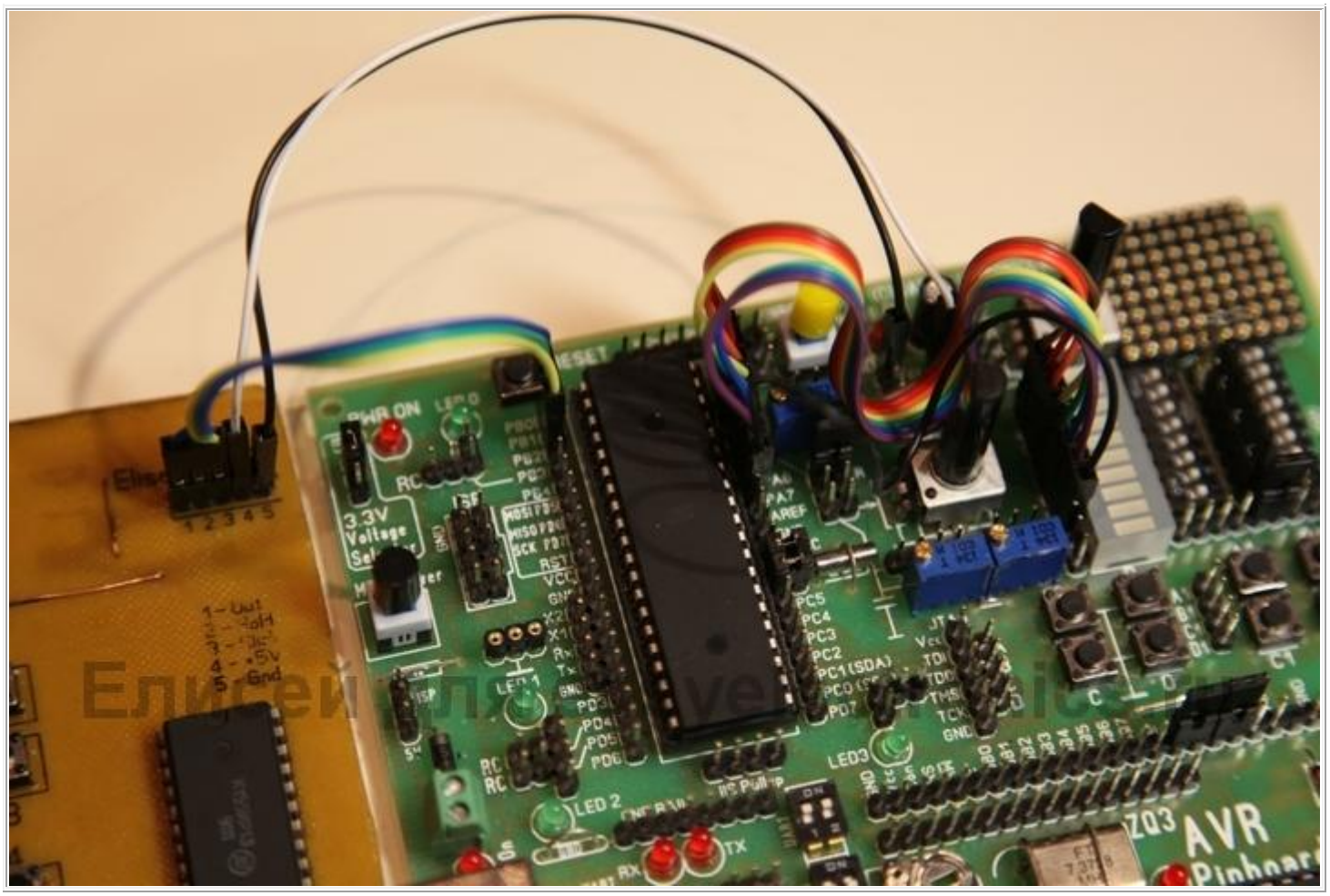
```
                .equ    BTN_DDR         =       DDRB
                .equ    BTN_PIN         =       PINB

                .equ    BTN_DATA_IN     =       0
                .equ    BTN_HOLD        =       1
                .equ    BTN_CLK         =       2

btn_start:      SBI     BTN_PORT,BTN_HOLD       ; Поднимаем S1

                SBI     BTN_PORT,BTN_CLK        ; кликаем
                CBI     BTN_PORT,BTN_CLK

                CBI     BTN_PORT,BTN_HOLD       ; опускаем S1

btn_again:      LDI     R17,0                   ; В этом регистре будет накапливаться наш
результат.
                                                ; неизвестно где он побывал, а нам
гарантированно
                                                ; нужен ноль.
                LDI     R16,8                   ; счетчик. Цикл будем проделывать 8 раз

btn_loop:       LSL     R17                     ; если мы проходим тут, первый раз, то данная
команда
                                                ; с нулем ничего не сделает, если же нет, то
двигаем все биты влево

                SBIC    BTN_PIN,BTN_DATA_IN     ; если к нам на вход пришла 1,
                INC     R17                     ; записываем 1 в самый младший разряд
регистра R17

                SBI     BTN_PORT,BTN_CLK        ; кликаем
                CBI     BTN_PORT,BTN_CLK

                DEC     R16                     ; уменьшаем счетчик
                BREQ    btn_exit                ; если счетчик досчитал до нуля, то переходим
выходим btn_exit
                Rjmp    btn_loop                ; иначе повторяем цикл, где первой же
командой сдвигаем все биты
                                                ; влево. Таким образом старые старшие байты
постепенно
                                                ; сдвигаются на свое место.

btn_exit:               RET
```

На данном этапе я сразу выделяю ее как функцию, чтобы далее было проще ее использовать.
Функцию написали, теперь создадим простейшую программу (это только для начала, развлечения впереди). Она в бесконечном цикле будет вызывать функцию сканирования клавиатуры, и полученный результат записывать в выходной порт PORTA, к которому мы подсоединим линейку светодиодов. Ничего сложного.

```
; Сначала инициализируем ноги контроллера:
        SBI     BTN_DDR,BTN_HOLD                ; выход HOLD
        SBI     BTN_DDR,BTN_CLK                 ; Выход CLK
        SBI     BTN_PORT,BTN_DATA_IN            ; вход DATA_IN

        OUTI    DDRA,0xFF                       ; Порт D ставим на выход. К нему будут
                                                ; подключаться светодиоды

;Затем создаем бесконечный цикл, в котором помещаем следующее:

main:   RCALL   btn_start                       ; вызываем функцию опроса клавиатуры
        COM     R17                             ; инвертируем пришедший байт
        OUT     PORTA,R17                       ; и записываем его в порт A
        RJMP    Main                            ; возвращаемся на начало
```

Инвертируем регистр R17 для нагдяности. Так как изначально бит нажатой кнопки у нас Ноль, а светодиод загорается при единице.

Компилируем и заливаем в замечательную отладочную плату PinBoard. Подключаем наш блок кнопок так, как показано на фотографии.



- 1 пин блока кнопок — PORTB.0
- 2 пин блока кнопок — PORTB.1
- 3 пин блока кнопок — PORTB.2
- 4 пин блока кнопок — +5 V
- 5 пин блока кнопок — Общий провод (Ground)
- PORTA (0..7) — линейка светодиодов.

Перезагружаем МК и наблюдаем следующую картину

При нажатии кнопки у нас загорается соответствующий светодиод. Причем возможен вариант одновременности нажания нескольких кнопок. А это потенциальный плюс, который, при желании, можно лихо использовать.

На этом конец первой части.

В следующей стетье я расскажу вам как увеличить число кнопок до 16 (24, 32 и т.д), как легко определять какая кнопка нажата в данный момент, и как на нажатие каждой кнопки прикрутить свою функцию.

В самом конце у нас получится примерно это

Все вопросы и пожелания оставляйте в комментариях.

- Проект для AVRStudio [2]

- Даташит для sn74198n [3]
- Печатная плата в формате Sprint Layout [4]
- Проект Proteus [5]

Смотри также статьи

- Матричная клавиатура [6]
- Сдвиговый регистр [7]
- Управление семисегментными индикаторами по одному проводу [8]

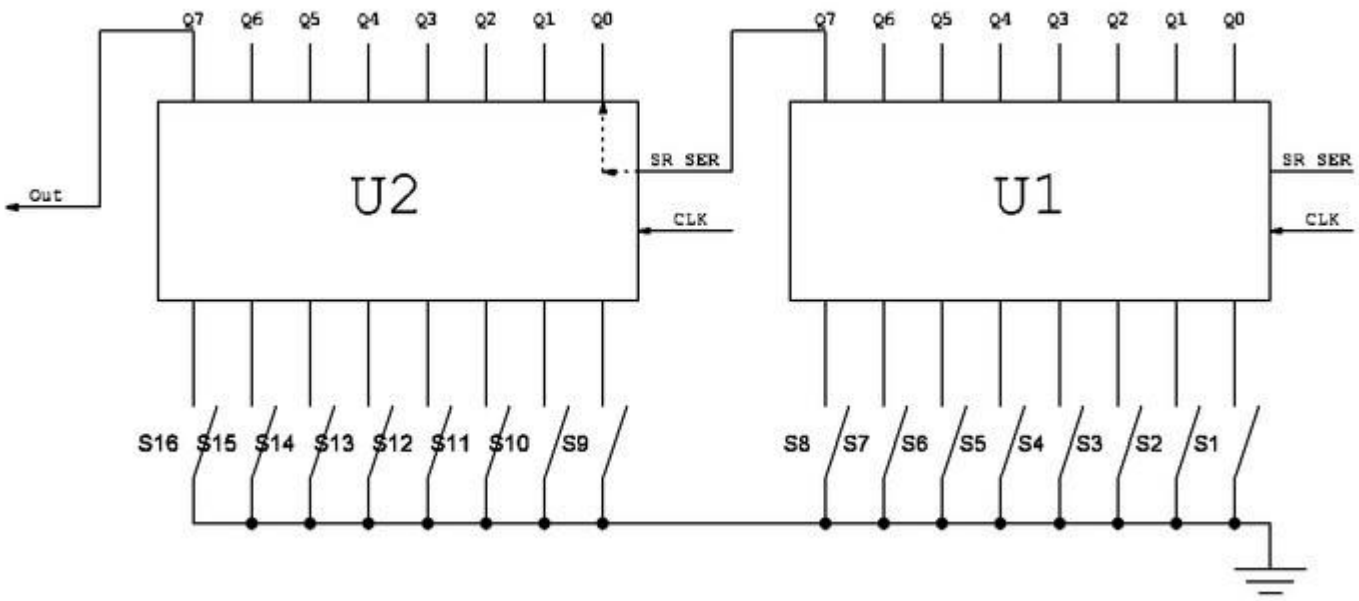
UPD
Продолжение банкета! Часть 2 [9]

Подключение клавиатуры к МК по трем проводам на сдвиговых регистрах. Часть 2. Буквенный ввод как на телефоне

Данная статья является продолжением предыдущей о подключении клавиатуры к МК с помощью трех сигнальных проводов [1]. В этой часте я расскажу вам о том, как увеличить число кнопок на клавиатуре до 16, опишу алгоритм обработки нажатий этих кнопок и то, как ассоциировать с каждой кнопкой свою функцию. Итак, приступим.

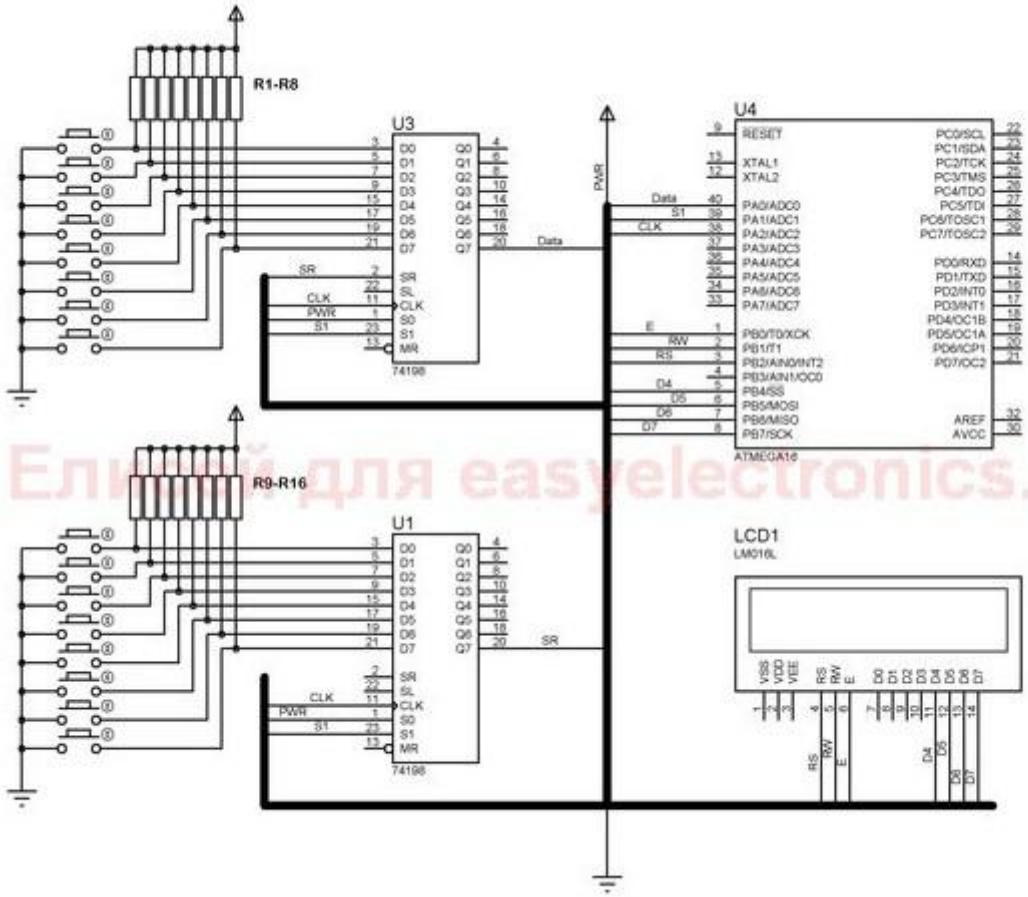
Аппаратная часть
Как вы уже догадались, чтобы подключить дополнительные кнопки к блоку клавиатуры, нужно добавить дополнительный сдвиговый регистр, который будет захватывать нажатия других восьми кнопок. Ниже приведена блок схема этой конструкции:

Рассмотрим режим работы, когда при каждом клике ногой CLK происходит сдвиг битов влево по направлению к старшему (S0 поднята, S1 опущена). Взглянем на сдвиговый регистр U1. При каждом дрыге ногой CLK, бит, который находится на выводе Qn, перемещается на вывод Qn+1, тоесть сдвигается в сторону старшего бита (влево). Но так как биту, который находится на ноге Q7 уже некуда сдвигаться, то он по идее должен бы был пропасть. Чтобы этого не произошло, мы посылаем его на следующий сдвиговй регистр U2, подключив Q7 регистра U1 к ноге SR SER регистра U2. Объясню, что же это за нога. В рассматриваемом нами режиме работы сдвигового регистра (S0 поднята, S1 опущена) биты смещаются в сторону старшего, а на место младшего становится бит, который в данный момент находится на ноге SR SER. Так как два наших сдвиговых регистра тактируются от одного источка (микроконтроллера), то бит на ноге Q7 сдвигового регистра U1, при сдвиге не теряется, а перескакивает на сдвиговый регистр U2, где продолжает свой путь в микроконтроллер.
Помимо SR SER, существует нога SL SER. Она обладает практически идентичными свойствами, за исключением того, что она используется при сдвигании регистров вправо, а не влево (режим, который мы не используем, S0 опущена, S1 поднята. В данном режиме биты будут двигаться по направлению к младшему байту, т.е вправо).



Таким образом, соеденив два сдвиговых регистра, мы по сути получаем один 16-ти битный, поведение которого абсолютно идентично 8-ми битному, за исключением того, что каждый раз нам необходимо считывать не 8, а 16 бит. Как я уже говорил в первой статье, время на сканирования такой клавиатуры возрастает примерно в 2 раза. Безболезненно к данной схеме можно добавлять все новые и новые сдвиговые регистры, подключая Q7 пин предыдущего к SR SER последующего. В данной статье мы ограничимся лишь двумя.

Ниже представлена схема данного устройства



[Тыц крупным планом](#) [2]

Схема упрощенная, показано только подключение клавиатуры и LCD. Питание и прочая обвязка контроллера как обычно.

Повторюсь, немаловажная деталь — подтягивающие резисторы R1 — R16. Если вы не знаете их назначения, прочитайте еще раз пункт «Описание кнопок» в первой части.

Далее переходим к написанию кода, который будет сканировать все 16 кнопок и что-то делать.

Програмная часть
За основу программы я возьму микроядро (простая операционная система), описанное на этом сайте. Подробности его установки и работы с ним вы можете прочитать в соответствующих статьях AVR курса [3], тут же я не буду подробно останавливаться на тонкостях его работы.

Начинаю программы я всегда с пустого шаблона, в котором данная ОС уже готова к работе, а нам лишь нужно добавить задачи и инициализировать их запуск. Так же у меня есть привычка разбивать программу на отдельные файлы, в каждом из которых находятся разные части нашей программы. Например инициализация, макросы, векторы прерываний, дополнительные функции и т.п. По ходу статьи я буду пояснять отдельные куски кода, показывать что откуда вышло, и для чего мы что-то делаем. Проект вцелом вы можете скачать по ссылке в конце статьи, и по нему отслеживать описание всех функций. В проекте весь код достаточно подробно прокоментирован.

Итак, у нас есть заготовка (ее вы тоже можете скачать вконце статьи), помимо нее мы будем использовать готовые библиотеки, работу которых я пояснять не буду, просто расскажу что они делают. Исходники вы можете посмотреть сами. Переносим все необходимые нам файлы в папку с проектом и инклудим их в соответствующих местах кода.
Файлы с макросами инклудятся в самом начале проекта, а файлы с кодом, я обычно инклудю вконце (см. проект)

- библиотека для работы с LCD (lcd4.asm, lcd4_macro.inc, от DI HALT'a) [4]
- Функция для прекодирования кирилических символов в ANSI кодировке, в символы, которые будут понятны дисплею. (ansi2lcd.asm) Данное решение встретил на форуме. Переписал его с Си на Асм и пользуюсь).
- Функция для сканирования нашей клавиатуры. Собственно ее я опишу в этой статье. Я ее вынес отдельным файлом, для удобства ее последующего использования (keyb_scan_init.asm, keyb_scan.asm)

Сканирование клавиатуры
Принцип сканирования клавиатуры я описал в предыдущей статье. В данном случае у нас будет небольшое отличие, т.к. нам нужно считать не 8, а 16 бит, т.е. 2 байта, со сдвиговых регистров.

Общий план действий

1. Устанавливаем бит T в регистре SREG. (Это пользовательский бит, который можно использовать для любых нужд. В нашем случае установленный бит будет означать, что мы считываем первый байт с нашей клавиатуры, если при проверке этот бит будет сброшен, то будем считать, что действие происходит со считыванием второго байта).
2. В цикле считываем 8 бит из сдвигового регистра.
3. Проверяем бит T:
    - Если он установлен, то мы только что считали первый байт, прячем его в закрома, сбрасываем бит T и возвращаемся на пункт 2.
    - Если он сброшен, то мы только что считали второй байт. Задача выполнена, выходим.

Код

```
1          .equ    BTN_PORT        =       PORTA
2          .equ    BTN_DDR         =       DDRA
3          .equ    BTN_PIN         =       PINA
4
5          .equ    BTN_DATA_IN     =       0
6          .equ    BTN_HOLD        =       1
7          .equ    BTN_CLK         =       2
8
9  btn_start:      SBI     BTN_PORT,BTN_HOLD       ; поднимаем S1
10
```

```
11                SBI     BTN_PORT,BTN_CLK       ; Кликаем
12                CBI     BTN_PORT,BTN_CLK
13
14                CBI     BTN_PORT,BTN_HOLD      ; опускаем S1
15
16                SET                            ; устанавливаем бит T в регистре SREG.
17                                               ; данный бит мы устанавливаем как флаг
18 того, что мы считываем первый байт.
19 btn_again:     LDI     R17,0                  ; в этом регистре будет накапливаться
20 результат. обнуляем его
21                LDI     R18,8                  ; счетчик. цикл будем проделывать 8 раз
22
23 btn_loop:      LSL     R17                    ; если мы проходим тут, первый раз, то
24 данная команда с нулем ничего не
25                                               ; сделает, если же нет, то двигаем все биты
26 влево
27
28                SBIC    BTN_PIN,BTN_DATA_IN    ; если к нам на вход пришла 1,
29                INC     R17                    ; записываем 1 в самый младший разряд
30 регистра R17
31
32                SBI     BTN_PORT,BTN_CLK       ; кликаем
33                CBI     BTN_PORT,BTN_CLK
34
35                DEC     R18                    ; уменьшаем счетчик
36                BREQ    btn_loop_end           ; если счетчик досчитал до нуля, то
37 переходим в btn_loop_end
38                Rjmp    btn_loop               ; иначе повторяем цикл, где первой же
39 командой сдвигаем все биты влево.
40                                               ;Таким образом старые старшие байты
41 постепенно сдвигаются на свое место.

  btn_loop_end:  BRTC    btn_exit               ; если бит T сброшен (а это значит, что мы
  уже приняли второй байт), то выходим из функции

                 CLT                            ; иначе сбрасываем бит T (это значит что мы
  закончили прием первого байта, и будем
                                                ; принимать второй
                 MOV     R16,R17                ; сохраняем первый принятый байт в регистре
  R16
                 RJMP    btn_again              ; и возвращаемся к считыванию байта
  btn_exit:      RET
```

Сохраняем данную функция в файл keyb_scan.asm и кидаем в папку с проектом.
Далее нам необходимо проинициализировать ноги контроллера. Это дело лучше автоматизировать, чтоб потом не заудмываться и не писать руками то, что можно не писать. Создадим файл keyb_scan_init.asm и напишем в нем следующее:

```
1                SBI     BTN_DDR,BTN_HOLD       ;выход HOLD
2                SBI     BTN_DDR,BTN_CLK        ;Выход CLK
3                SBI     BTN_PORT,BTN_DATA_IN   ;вход DATA_IN
```

Этот файл просто подключаем к проекту в разделе инициализации, ничего в нем не меняя.

```
         .include        "init.asm"             ; в данном файле хранится общая
1 инициализация
2         .include        "keyb_scan_init.asm"   ; инициализация ног для сканирования
  клавиатуры
```

Итак, функция сканирования клавиатуры у нас готова, далее нужно обрабатывать приходящие данные. Но прежде я добавлю две задачи, которые будут служить индикацией работы нашей ОС. Ибо в процессе отладки кода, контроллер может уходить в ребут, и дабы это сразу замечать и не тратить время на локализацию проблемы, я

делаю моргалку диодом. Диод моргает — ОС работает.
Создаем две задачи. Одна зажигает диод, другая гасит. Причем обе вызывают друг друга с задержкой 500 мс.

```
1 SysLedOn:       SetTimerTask    TS_SysLedOff,500
2                 SBI     PORTD,5
3                 RET
4 ;-----------------------------------------------------------------------------
5 SysLedOff:      SetTimerTask TS_SysLedOn,500
6                 CBI     PORTD,5
7                 RET
```

И запустим их во время старта в области Background

```
1 RCALL   SysLedOn
```

Я не буду полностью описывать как добавить задачу в микроядро. Это достаточно подробно описано в соответствующих статьях, ссылки на которые я дал выше.

Далее перейдем в сканированию клавиатуры. Создадим задачу KeyScan и запустим ее в области Background

```
1 Background:     RCALL   SysLedOn
2                 RCALL   KeyScan
```

Функция KeyScan будет обрабатывать два байта последовательно, пришедших с клавиатуры, используя аналогичный метод с битом T. Так же для перехода по необходимым нам функнциям в зависимости от нажатой кнопки, мы будем использовать таблицу с адресами переходов.

```
1 Code_Table:     .dw     Key1,   Key2,   Key3,   Key4,   Key5,   Key6,   Key7,   Key8
2 Code_Table2:    .dw     Key9,   Key10,  Key11,  Key12,  Key13,  Key14,  Key15,  Key16
```

Это две таблицы, одна для клавиш с 1 по 8, другая — с 9 по 16. В ней последовательно расположены адреса на функции, которые мы будем выполнять в зависимости от того, какая кнопка нажата.
Для этого мы заранее загрузим адрес начала таблицы в регистровую пару Z, и затем, вычислив, какая же по счету кнопка была нажата, прибавим это смещение к адресу начала таблицы. Получим адрес с ячейкой, в которой содержится адрес функции, которую нужно выполнить. Звучит немного сложно, но на самом деле, все достаточно просто и понятно. Главно вчитаться в предыдущее предложение.

Обратите внимание, что в таблице адреса занимают по 2 байта, а смещение у нас будет увеличиваться на один, поэтому нам необходимо будет умножить смещение на два. Вы увидете это в коде. Знайте, что это из за того, что адрес занимает два байта, и нам нужно перепрыгнуть через оба.

Другая особенность:
При отсутствии нажатий с клавиатуры приходит 0b11111111. Тоесть ненажатая кнопка — высокий уровень. К примеру нажмем кнопку 3, и к нам придет число 0b11110111 (соответствующий бит сброшен). Поэтому выведем алгоритм: пришедший байт мы сначала будем сравнивать с маской 0b11111110, потом с 0b11111101, затем 0b11111011 и т.д. Мы просто будем в цикле сдвигать биты в маске влево, каждый раз сверяя ее с пришедшим байтом и увеличивая счетчик. В тот момент, когда будет совпадение — в счетчике будет номер нажатой кнопки. Что нам собственно и требуется.
В функции будет использоваться один байт из оперативной памяти.

```
1 ; RAM =====================================
2                 .DSEG
3 KeyFlag:        .byte   1
```

В нем будет храниться последнее зарегестрированное нажатие. После записи туда номера кнопки, мы будем ставить функцию очистки этого байта через 200 мс. Это сделано для того, чтобы повторная обработка нажатия этой кнопки не производилась ранее этого времени, т.е. только до того, как KeyFlag будет сброшен. Это делается для защиты от случайных двойных нажатий.
Итак, код сканирования:

```
KeyScan:        SetTimerTask   TS_KeyScan,50

                RCALL   btn_start       ; сканируем клавиатуру. результат приходит в
регистрах R16 и R17

                SET                     ; ставим флаг T в регистре SREG. Он будет
означать, что мы обрабатываем первый принятный с клавиатуры байт

                LDI     ZL,low(Code_Table*2)    ; берем адрес первой таблицы с переходами
                                        ; (для кнопок 1-8)
                LDI     ZH,high(Code_Table*2)

KS_loop:        CPI     R16,0xFF                ; если байт равен 0xFF, то нажатия не было,
                BREQ    KS_loop_exit            ; переходим на обработку следующего байта с
клавиатуры

                LDS     R18,KeyFlag             ; берем последнее зарегестрированное
нажатие
                CP      R16, R18                ; сравниваем с текущим
                BREQ    KS_loop_exit            ; если одинаковы, переходим на обработку
следующего
                                                ; байта с клавиатуры
                STS     KeyFlag,R16             ; иначе сохраняем в RAM текущее нажатие
                                                ; как последнее зарегестрированное

                PUSH    R16
                PUSH    R17

                SetTimerTask    TS_ClearFlag, 200       ; ставим на запуск через 200 мс
функцию очистки
                                                ; последнего зарегестрированного нажатия

                POP     R17                     ; данная функция использует R16 и
                POP     R16                     ; R17, поэтому сохраняем их в стеке

                RJMP    KS_got_smth             ; если мы дошли до этого места, то у нас
                                                ; есть нажатие, которое нужно обработать.
идем на обработку

KS_loop_exit:   BRTC    KS_exit                 ; проверяем флаг T в регистре SREG. Если он
                                                ; не сброшен, а значит мы считали только
                                                ; один байт с клавиатуры, то идем
                                                ; дальше, иначе выходим

                CLT                             ; сбрасываем флаг T.Это означает что мы
считали
                                                ; первый байт с клавиатуры, и готовы
                                                ; ко второму.

                LDI     ZL,low(Code_Table2*2)           ; берем адрес второй таблицы с
                LDI     ZH,high(Code_Table2*2)          ; переходами (для кнопок 9-16)
                MOV     R16,R17                 ; второй принятый байт перекидываем в R16
                RJMP    KS_loop                 ; и возвращаемся в цикл

; тут мы оказываемся, когда нам нужно обработать нажатие.
;
KS_got_smth:    CLR     R18                     ; R18 будет счетчиком. Нужно сравнить 8
возможных состояний пришедшего байта,
                                                ;  поэтому будем считать до 8
                LDI     R19,0b11111110          ; первоначальная маска для сравнения ее с
пришедшим битом, и дальнейшего сдвигания влево

KS_loop2:       CP      R16,R19         ; сравниваем маску с пришедшим байтом
```

```
64                BREQ    KS_equal        ; если равны, то переходим на действие
65
66                INC     R18             ; иначе увеличиваем счетчик
67                CPI     R18,8           ; сравниваем его с восьмеркой
68                BREQ    KS_exit         ; если досчитали до 8, то выходим
69
70                SEC                     ; тут двигаем нашу маску влево. так как младшие
71 байты нам нужно заполнять
72                                        ; единицами, а функция ROL устанавливаем эту
73 единицу только при наличии флага C,
74                                        ;то устанавливаем этот флаг
75                ROL     R19             ; двигаем биты в маске
76                RJMP    KS_loop2        ; и переходим опять на цикл
77
78 KS_equal:      LSL     R18             ; R18 хранится число, до которого мы успели
79 досчитать,
80                                        ; пока ждали совпадения байта
81                                        ; с клавиатуры с маской.В нем по сути находится
   номер нажатой кнопки.
                                          ; умножаем его на 2, так как в талице переходов
   адреса
                                          ; хранятся по 2 байта
                  ADD     ZL,R18          ; складываем смещение с заранее сохраненным
   адресом таблицы переходов
                  ADC     ZH,R0           ; в R0 я всегда храню ноль

                  LPM     R16,Z+          ;загружаю необходимый адрес из таблицы
                  LPM     R17,Z

                  MOV     ZL,R16          ; перекидываем его в адресный регистр Z
                  MOV     ZH,R17

                  ICALL                   ; и вызываем функцию по этому адресу
   KS_exit:       RET
```

Вместо ICALL можно в данном случае применить IJMP будет примерно тот же эффект, но выход из KeyScan будет через RET в вызваной функции. Не так очевидно, зато сэкономим два байта стека :) Формально это можно представить как то, что наша функция KeyScan это этакий многозадый кащей. Вошли в одну голову, а вывались через одну из задниц определенных нажатием клавиши.

Наверняка вы заметили следующую строчку:

```
1 SetTimerTask    TS_ClearFlag,200
```

Данный макрос устанавливает на выполнение функцию ClearFlag через 200 мс. Данная функция должна удалить из ячейки KeyFlag в оперативной памяти информацию о прошлом нажатии. Так как при отсутствии нажатий с клавиатуры приходит байт 0b11111111, то в функции ClearFlag и будем записывать в ячейку KeyFlag это число:

```
1 ClearFlag:      SER     R16             ; R16 = 0xFF
2                 STS     KeyFlag,R16     ; сохраняем это в RAM
3                 RET
```

Теперь рассмотрим таблицу с адресами переходов повнимательнее.

```
1 Code_Table:     .dw     Key1,   Key2,   Key3,   Key4,   Key5,   Key6,   Key7,   Key8
```

Code_Table — адрес начала таблицы. Прибавляя к этому адресу необходимое нам смещение, мы будем получать адрес ячейки, в которой хранится адрес перехода (Key1, Key2, Key3 и т.д) на нужную нам функцию. Директива .dw означает что для каждого элемента, описанного далее в строке выделяется по 2 байта. Выделяем столько, ибо адреса у нас двухбайтовые.

Итак, переходы на нужные нам функции при нажатии клавишь у нас есть. Теперь, чтоб выполнить какой-либо код, при нажатии на кнопку 1, нам нужно в любом месте программы добавить следующую функцию:

```
  Key1:   LDI     R16,0x02        ; просто какой-то случайный код. не несет в себе смысла.
1                                 ; Тут вы подставите то, что нужно будет выполнить вам при
2 нажатии на кнопку 1
3         LDI     R17,0x03
4         SUB     R16,R17
5         RET                     ; обязательно выход из этой функции по RET, иначе будет
  переполнение стека
```

Тут собственно можно было бы и остановиться. Я рассказал принцип действия данной клавиатуры, рассказал о функциях сканирования и перехода по заданным адресам в зависимости от нажатой клавиши, но все же я расскажу вам, как данную клавиатуру можно применить. Сделаем ввод текста на дисплей. Так как кнопок у нас немного, то полноразмерную QWERTY клаву сделать не получится, поэтому обойдемся тем что есть. Будем делать ввод текста как на телефоне. T9 я реализовывать не буду, ибо это достаточно трудоемко в качестве примера. Поэтому на каждую кнопку прикрутим по 4 символа, которые будут поочередно выводиться на дисплей при каждом нажатии. При задержке нажатия на определенное время (например 1 секунда) происходит сдвиг курсора. Так же реализуем команды пробел, стереть символ, очистить дисплей, и перемещение курсора влево, вправо, вверх, вниз.

Начнем с букв и символов. Как я уже говорил, к каждой кнопке мы прикрутим по 4 символа. Для этого создадим таблицы этих символов, по которым мы будем их перебирать:

```
1 Letter_K_Table1:      .db     0x2E,0x2C,0x3F,0x21,0, 0            ;"., ",", "?", "!"
2 Letter_K_Table2:      .db     0xE0,0xE1,0xE2,0xE3,0, 0            ;а, б, в, г
3 Letter_K_Table3:      .db     0xE4,0xE5,0xE6,0xE7,0, 0            ;д, е, ж, з
4 Letter_K_Table4:      .db     0xE8,0xE9,0xEA,0xEB,0, 0            ;и, й, к, л
5 Letter_K_Table5:      .db     0xEC,0xED,0xEE,0xEF,0, 0            ;м, н, о, п
6 Letter_K_Table6:      .db     0xf0,0xf1,0xf2,0xf3,0, 0            ;р, с, т, у
7 Letter_K_Table7:      .db     0xf4,0xf5,0xf6,0xf7,0, 0            ;ф, х, ц, ч
8 Letter_K_Table8:      .db     0xf8,0xf9,0xfa,0xfb,0, 0            ;ш, щ, ъ, ы
9 Letter_K_Table9:      .db     0xfc,0xfd,0xfe,0xff,0, 0            ;ь, э, ю, я
```

Тут последовательно забиты строчные буквы кирилицы в ANSI кодировке. каждая таблица заканчивается нулем. так как в таблице должно быть четное количество байт, то я добавил еще по нулю вконце. Это немного расточительно с точки зрения использования памяти, но увы и ах — адресация у нас тут словами.

Очевидно предположить, что при нажатии любой из девяти кнопок будет выполняться один и тот же код, но будут использоваться разные данные. А это значит, что мы создадим функцию, в которую будут передаваться номер нажатой кнопки и адрес начала нашей таблицы с символами для этой кнопки. Таблицы были только что описаны выше.

В обработчике нажатия первой кнопки Key1 запишем следующий код:

```
  ; символы "." "," "?" "!"
  key1:           LDI     ZL,low(Letter_K_Table1*2)       ; загружаем в Z адрес начала таблицы
1                 LDI     ZH,high(Letter_K_Table1*2)      ; с символами, принадлежащей первой
2 кнопке
3                 LDI     R16,1                           ; загружаем в R16 номер нажатой
4 кнопки
5                 RCALL   lcd_write_1                     ; вызов функции вывода символа в
6 видеопамять
                  RET
```

Аналогичные действия проделываем с восемью другими функциями. Код приводить не буду, если нужно — все есть в проекте.

Как вы уже догалались, все действие будет происходить в функции lcd_write_l. Она будет сверять пришедшее нажатие с предыдущим, и в зависимости от результата брать следующий символ из таблицы символов и помещать его на место последней буквы (если кнопка нажата повторно), либо записывать первый символ из таблицы в

новую ячейку видеопамяти (если нажата новая кнопка). Также будет использоваться макрос установки задачи по отчистке последнего нажатия кнопки с отсрочкой на определенное время.
Принцип действия практически аналогичный тому, который использовался для защиты от случайных повторов, при сканировании клавиатуры, только задержка по времени больше.

Символы в LCD мы будем записывать не напрямую, а через промежуточную видеопамять, которая будет находиться в оперативной памяти *(Хехехе дается мне на это повлиял алгоритм демопроги, что шел в документации к Pinboard прим.* ***DI HALT*** *;))* . Это сделано для удобства последующего наращивания функционала программы. Набранный текст будет проще сохранять, обрабатывать, посылать на ПК и т.д. Позднее мы создадим задачу обновления дисплея, которая, периодически запускаясь, будет записывать символы из видеопамяти в дисплей. Получается такого рода отвязка основной логики программы от железа.
При необходимости, с легкостью можно будет применить любой другой дисплей, переписал лишь только функцию его обновления. Данную абстракцию логики программы от железа я произвожу в учебных целях. Пусть даже данное решение излишне для нашего задания, но правильно написанная программа впоследствии ползволяет сэкономить кучу времени себе и другим программистам. Поэтому лучше сразу привыкать писать правильно. (Как писать правильно, а как нет, это лишь мое сугубо личное мнение. У кого-то оно может отличаться. Я не навязываю свою точку зрения, я рассказываю то, что знаю сам).

Создаем ячейки для видеопамяти в RAM и кое-какие переменные:

```
1                .equ    LCD_MEM_WIDTH = 32     ; размер памяти LCD. у меня дисплей 2 строки
2 по 16 символов.
3 LCDMemory:     .byte   LCD_MEM_WIDTH
4
5 PushCount:     .byte   1              ; счетчик нажатий на кнопку
6 KeyFlagLong:   .byte   1              ; тут хранится номер последней нажатой кнопки
  CurrentPos:    .byte   1              ; текущее положение курсора
```

Далее, привожу код всей функции.

```
1                .equ    delay   =       1000    ; задержка перед сдвигом курсора
2
3  lcd_write_1:  LDS     R17,KeyFlagLong         ; загружаем номер последней нажатой кнопки
4                CP      R16,R17                 ; сравниваем его с текущей нажатой кнопкой
5                BREQ    lwl_match
6
7  ;---нажата новая кнопка---
8  lwl_not_match: STS    KeyFlagLong,R16         ; была нажата другая кнопка. сохраняем ее
9  номер в RAM
10               CLR     R17
11               STS     PushCount,R17           ; и обнуляем счетчик нажатий кнопки, ибо
12 эту кнопку мы нажали первый раз
13               RJMP    lwl_action
14
15 ;---повторно нажата кнопка---
16 lwl_match:    LDS     R17,PushCount           ; если же была нажата кнопка повторно
17               INC     R17                     ; увеличиваем счетчик нажатий
18
19               LDS     R18,CurrentPos
20               DEC     R18                     ; сдвигаем текущее положение курсора
21                                               ; влево. так как нам необходимо будет
22                                               ; заново переписать букву на прежнем месте
23               STS     CurrentPos,R18
24
25               PUSH    R17                     ; макрос SetTimerTask использует регистр
26 R17, поэтому заранее сохраняем его в стеке
27
28               SetTimerTask   TS_Reset_KeyFlagLong,delay    ; ставим задачу отчистки
29 номера
30                                               ; о текущей кнопки.по истечению
31                                               ; этого времени мы сможет повторно
32                                               ; одной кнопкой вывести вторую букву
```

```
33                POP     R17
34
35 lwl_action:    ADD     ZL,R17          ;прибавляем смещение к адресу таблицы с ANSI
36                                        ; символами, принадлежащими данной кнопке
37                ADC     ZH,R0
38
39                LPM     R16,Z           ; загружаем нужный нам символ из таблицы
40
41                CPI     R16,0           ; проверка на ноль. Если ноль - то конец таблицы
42                BRNE    lwl_next_act    ; если не конец таблицы, то продолжаем действие
43 переходом на next_act
44
45                SUB     ZL,R17          ; иначе нам нужно вернуться на начало таблицы,
46                SBCI    ZH,0            ; поэтому обратно вычитаем смещение из адреса
47 нашей таблицы
48                CLR     R17             ; в R17 у нас лежит счетчик нажатий. Обнуляем его.
49                RJMP    lwl_action      ; и повторяем все действие заново. но как будто
50 это наше первое нажатие
51                                        ; на данную кнопку
52
53 lwl_next_act:  STS     PushCount,R17   ; прямчем в RAM счетчик нажатий
54
55                RCALL   ansi2lcd        ; преобразование ANSI в кодировку, пригодную для
56 LCD.
57                                        ; Вход и выход - R16. изменяет регистр R17
58
59 lwl_wr_mem:    LDS     R17,CurrentPos  ; загуржаем текущее положение курсора
60
61                LDI     ZL,low(LCDMemory*2)     ; загружаем адрес таблицы видеопамяти
62                LDI     ZH,high(LCDMemory*2)
63
64                ADD     ZL,R17          ; складываем смещение (положение курсора) с
65 началом таблицы
66                ADC     ZH,R0           ; R0 я держу всегда нулем
67
68                ST      Z,R16           ; сохраняем символ в видеопамяти

                  INC     R17             ; увеличиваем на 1 текущее положение

                  CPI     R17,LCD_MEM_WIDTH       ; сравниваем, достигло ли текущее положение
  конца памяти LCD
                  BRNE    lwl_not_end
                  CLR     R17             ; если да, обнуляем текущее положение

  lwl_not_end:    STS     CurrentPos,R17  ; и сохраняем текущее положение в RAM
                  RET
```

RCALL ansi2lcd — данная строчка вызывает функцию преобразования ANSI символа в кодировку, понятную LCD на базе HD44780. Так как по умолчанию эти дисплеи плохо дружат с кирилицей, приходится немного извращаться, чтоб корректно выводить кирилические символы. Принцип действия данной функции я описывать не буду, можете самостоятельно подсмотреть код в файле ansi2lcd.asm. Скажу лишь, что символ посылаем в регистре R16, и получаем оттуда же. Данная функция также изменяет регистр R17, будьте аккуратны, не оставляйте в нем ничего нужного.

Вообщем, запись необходимого символа в видеопамять у нас реализована. Перейдем к функции отрисовки дисплея из видеопамяти. Она будет в цикле поочередно брать символы из видеопамяти и посылать их в LCD. По сути ничего сложного. Единственно надо будет отследить, когда курсор достигнет конца первой строки, затем перевести его на вторую. Иначе символы запишутся не в видимую часть дисплея. Подробнее об видимых и невидимых областях памяти дисплея можно прочитать в это статье http://easyelectronics.ru/avr-uchebnyj-kurs-podklyuchenie-k-avr-lcd-displeya-hd44780.html

Создадим новую задачу ОС и назовем ее LCD_Reflesh. Поставим ее на первоначальный запуск в области Background

```
1 Background:     RCALL   SysLedOn
2                 RCALL   KeyScan
3                 RCALL   LCD_Reflesh
```

и напишем саму функцию:

```
   LCD_Reflesh:   SetTimerTask   TS_LCD_Reflesh,100     ; запускаем обновление дисплея
   каждый 100 мс
1
2                 LDI     ZL,low(LCDMemory*2)    ; грузим в Z адрес видеопамяти
3                 LDI     ZH,high(LCDMemory*2)
4
5                 LCD_COORD       0,0             ; устанавливаем текуюю координату курсора в
6  LCD в самое начало
7                 LDI     R18,LCD_MEM_WIDTH       ; грузим в R18 длину видеопамяти. это будет
8  нас счетчик
9
10 lcd_loop:      LD      R17,Z+          ; цикл. тут мы берем из видеопамяти один символ в
11 регистр R17
12                RCALL   DATA_WR         ; и записываем его в LCD.
13
14                DEC     R18             ; уменьшаем счетчик
15                BREQ    lcd_exit        ; если достигли конца видеопамяти - выходим
16
17                CPI     R18,LCD_MEM_WIDTH/2     ; достигли ли конца первой строки?
18                brne    lcd_next
19
20                LCD_COORD       0,1     ; если да, устанавливаем текущую координату
21                                        ; курсора в LCD на вторую строчку
22
   lcd_next:      RJMP    lcd_loop        ; и продолжаем цикл
   lcd_exit:      RET
```

Можно считать что минимальный рабочий код написал. Компилируем и прошиваем. Работать будут кнопки с буквами и символами. Подключаем к микроконтроллеру наш блок клавиатуры и LCD дисплей.

Клавиатура:

- 1 пин блока кнопок — PORTA.0
- 2 пин блока кнопок — PORTA.1
- 3 пин блока кнопок — PORTA.2
- 4 пин блока кнопок — +5 V
- 5 пин блока кнопок — Общий провод (Ground)

LCD дисплей:

- Пин E — PORTB.0
- Пин RW — PORTB.1
- Пин RS — PORTB.2
- Пин Data.4 — PORTB.4
- Пин Data.5 — PORTB.5
- Пин Data.6 — PORTB.6
- Пин Data.7 — PORTB.7

О распиновке LCD дисплея очень много информации в интернете. Гуглите «подключение WH1602”. Подробности о том, как подключить LCD дисплей к плате PinBoard — смотрите в инструкции к ней.
Запускаем демоплату и наблюдаем следующую картину.

Сначала удивляемся. С первого взгляда ничего не работает и на экране каракули. Но, понажимав на кнопки, можно убедиться, что символы последовательно выводятся на экран, заменяя собой эти самые каракули. Значит проблема в начальной инициализации дисплея, а точнее в видео памяти. С самого начала видеопамять заполнена

нулями (Мы же в нее при инициализации ничего не записываем).
И далее эти нули посылаются в дисплей, где они превращаются в крякозябру. А нам нужна пустота, символ пробела (не знаю как его еще назвать=)).

Немного погуглив, находим таблицу ANSI символов, где беглым взгялдом находим нужный нам символ — пустоту. Его код — 0×20. Отлично, при инициализации контроллера заполним ячейки видеопамяти этими числами. Создаем функцию очистки видеопамяти. Выглядит она следующим образом:

```
   LCD_Clear:     LDI     ZL,low(LCDMemory*2)    ; загружаем в Z адрес таблицы видеопамяти
1                 LDI     ZH,high(LCDMemory*2)
2
3                 LDI     R16,0x20               ; 0x20 - код пустого символа.
4                                                ; при отчистке заполняем им всю видеопамять
5                 LDI     R17,LCD_MEM_WIDTH      ; в счетчик кладем длину видеопамяти
6
7  LCl_loop:      ST      Z+,R16                 ; записываем 0x20 в текущую ячейку памяти
8                 DEC     R17                    ; уменьшаем счетчик
9                 BRNE    LCl_loop               ; если он еще не достиг нуля, повторяем
10 цикл
11
12                STS     CurrentPos,R0          ; текущую позиция курсора устанавливаем в
13 ноль
                  RET
```

Далее переходим в область инициализации (init.asm) нашего проекта и добавляем

```
1                RCALL   LCD_Clear       ; отчищаем память дисплея
```

Компилируем, прошиваем и убеждаемся что теперь все работает корректно.
Далее по заданию нам нужно реализовать кнопки: стереть текущий символ, отчистить дисплей, навигация вверх, вниз, влево, вправо, пробел.

Стереть
Данное действие прикручиваем к кнопке 4. А действие тут простое: уменьшаем текущее положение курсора на 1 (ибо стираем предыдущий символ), прибавляем текущее положение к адресу начала таблицы видео памяти (получаем адрес ячейки с текущим символом), и записываем туда число 0×20 (символ пустоты, пробела). Нам нужно будет произвести одну проверку: находясь в самой первой ячейке, мы не можем использовать данную функцию по понятным причинам.

```
   ;стереть
1  key4:          LDS     R17,CurrentPos         ; загружаем текущее положение курсора
2                 TST     R17                    ; проверяем его
3                 BREQ    key4_exit              ; если оно ноль, то выходим
4
5                 DEC     R17                    ; уменьшаем текущее положение, т.е.
6                                                ; переходим на символ, который нужно
7  стереть
8                 STS     CurrentPos,R17         ; сохраняем данное значение в RAM
9
10                LDI     ZL,low(LCDMemory*2)    ; грузим адрес таблицы видеопамяти
11                LDI     ZH,high(LCDMemory*2)
12
13                ADD     ZL,R17                 ; складываем смещение (положение курсора) с
14 началом таблицы
15                ADC     ZH,R0                  ; R0 я держу всегда нулем
16
17                LDI     R16,0x20               ; загружаем в R16 число 0x20.
18                                               ; В LCD оно означает пустое место, пробел.
19                ST      Z,R16                  ; сохраняем символ в видеопамять
   key4_exit:     RET
```

Отчистка экрана
Проще некуда. Вызываем функцию LCD_Clear

```
; Отчистка экрана
key8:           RCALL   LCD_Clear
                RET
```

Навигация
Все махинации сводятся к изменению текущего положения, но при наличии проверок: находясь на первой строке, мы не можем перейти вверх, находясь на нижней — вниз.

```
; курсор вверх
key12:          LDS     R16,CurrentPos          ; загружаем текущее положение курсора
                CPI     R16,LCD_MEM_WIDTH/2+1   ; проверяем, на какой строке он находится
                BRMI    key12_exit              ; если на первой, то выходим

                SUBI    R16,LCD_MEM_WIDTH/2     ; иначе переходим на строку выше

                STS     CurrentPos,R16          ; сохраняем текущее положение в RAM

                STS     KeyFlagLong,R0          ; обнуляем информацию о последней нажатой
кнопки
                STS     PushCount,R0            ; и счетчик нажатий

key12_exit:     RET


; курсор вниз
key16:          LDS     R16,CurrentPos          ; загружаем текущее положение курсора
                CPI     R16,LCD_MEM_WIDTH/2     ; проверяем, на какой строке он находится
                BRPL    key16_exit              ; если на второй, то выходим

                LDI     R17,LCD_MEM_WIDTH/2     ; иначе переходим на строку ниже
                ADD     R16,R17

                STS     CurrentPos,R16          ; сохраняем текущее положение в RAM

                STS     KeyFlagLong,R0          ; обнуляем информацию о последней нажатой
кнопки
                STS     PushCount,R0            ; и счетчик нажатий

key16_exit:     RET


; курсор влево
key13:          LDS     R16,CurrentPos          ; загружаем текущее положение курсора
                DEC     R16                     ; уменьшаем его на единицу
                STS     CurrentPos,R16          ; и сохраняем где был

                STS     KeyFlagLong,R0          ; обнуляем информацию о последней нажатой
кнопки
                STS     PushCount,R0            ; и счетчик нажатий

key13_exit:     RET


; курсор вправо
key15:          LDS     R16,CurrentPos          ; загружаем текущее положение курсора
                INC     R16                     ; увеличиваем его на единицу
                STS     CurrentPos,R16          ; и сохраняем где был

                STS     KeyFlagLong,R0          ; обнуляем информацию о последней нажатой
```

```
кнопки
                STS     PushCount,R0            ; и счетчик нажатий
key15_exit:     RET
```

Ну и самое главное — клавиша пробела:

```
;пробел
key14:          LDS     R17,CurrentPos          ; загружаем текущее положение курсора
                INC     R17                     ; увеличиваем его на единицу
                STS     CurrentPos,R17          ; и сохраняем где был

                LDI     ZL,low(LCDMemory*2)     ; грузим адрес таблицы видеопамяти
                LDI     ZH,high(LCDMemory*2)

                ADD     ZL,R17                  ; складываем смещение (положение курсора) с
началом таблицы
                ADC     ZH,R0                   ; R0 я держу всегда нулем

                LDI     R16,0x20                ; загружаем в R16 число. В LCD оно означает
пустое место, пробел.
                ST      Z,R16                   ; сохраняем символ в видеопамяти

                STS     KeyFlagLong,R0          ; обнуляем информацию о последней нажатой
кнопки
                STS     PushCount,R0            ; и счетчик нажатий
                RET
```

Снова компилируем, прошиваем программу в контроллер и запускаем. Убеждаемся что все кнопки работают и выполняют свою функцию.

Про антидребезг и защиту от помех
В коментариях к прошлой статье поднялся вопрос об устранении дребезга и защиты такой клавиатуры от помех. Расскажу о том как он реализован тут, и о различных аппаратных и програмных способах его реализаци. Благодаря применению микроядра (операционная система), можно легко организовать програмный антидребезг, путем запуска сканирования клавиатуры через определенные промежутки времени, большие чем длительность дребезга. Таким образом, при улавливании нажатия, мы уходим на его обработку и возвращаемся к сканированию клавиатуры спустя какое-то время, когда дребезг от данного нажатия уже кончился. Таким образом отсутствует возможность из-за дребезга посчитать одно нажатие дважды. Но тут вылазиет другая проблема: защита от помех.

DVF января 4, 2011 at 5:51
У применения как программного, так и аппаратного антидребезга есть еще одна важная сторона — это испытание системы на помехоустойчивость. Ведь, если не дребезг, то помеху можно принять за “сработку” в момент сканирования. Для любительской практики это может и не важно, а для тех, кто профессионально занят в разработках — напротив. Подтянутая линия входа от кнопки потенциальный источник сбоя, нежели выход логики.

Действительно, случайная помеха может вызвать ложное срабатывание. Способ борьбы с этим очевиден: производить несколько сканирований клавиатуры, и при положительном результате каждого сканирования, считать что было нажатие. Это, естественно, увеличивает сложность кода и процессорное время, затрачиваемое на обработку нажатий. Но, как правило, время сканирования много меньше времени програмного антидребезга, поэтому дополнительная обработка не влечет за собой больших трудностей.

Так же существует варианты аппаратной защиты от дребезга с помощью конденсаторов, триггеров и др. Но это уже совсем другая статья.
Всем спасибо за внимание, вопросы и пожелания оставляйте в комментариях.

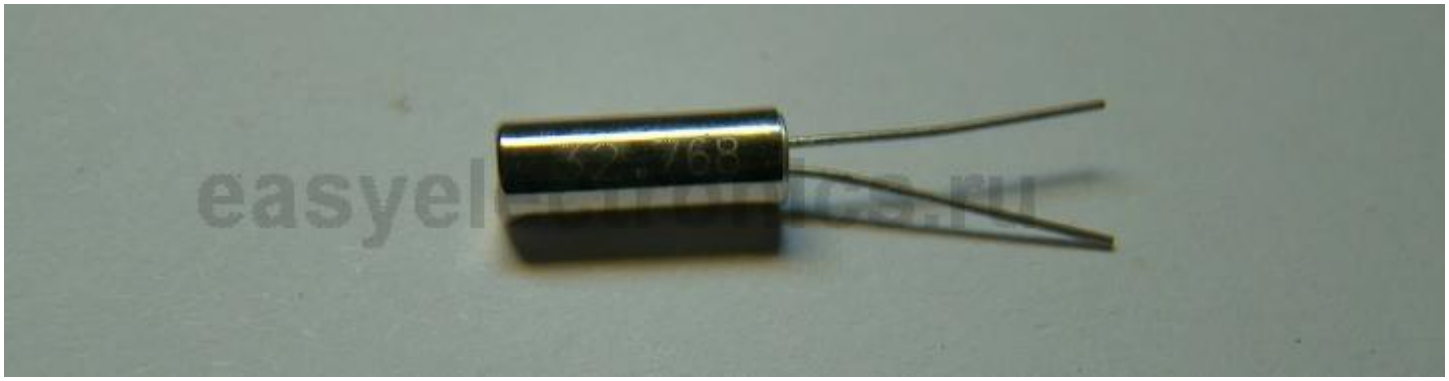
Файлы:

- Проект для AVR Studio **buttons_2.rar** [5]
- Шаблон микроядрая для AVR Studio **template_RTOS.rar** [6]

AVR. Учебный Курс. Асинхронный режим таймера

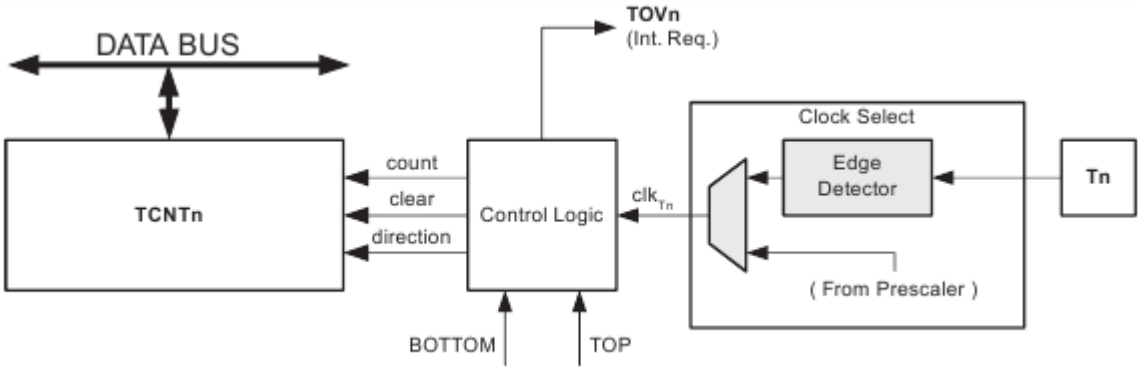
Иногда полезно иметь в системе часы отсчитывающие время в секундах, да еще с высокой точностью. Часто для этих целей применяют специальные микросехмы RTC (Real Time Clock) вроде PCF8583 [1]. Вот только это дополнительный корпус, да и стоит она порой как сам МК, хотя можно обойтись и без нее. Тем более, что многие МК имеют встроенный блок RTC. В AVR его правда нет, но там есть асинхронный таймер, служащий полуфабрикатом для изготовления часиков.

Первым делом нам нужен часовой кварц на 32768Герц.

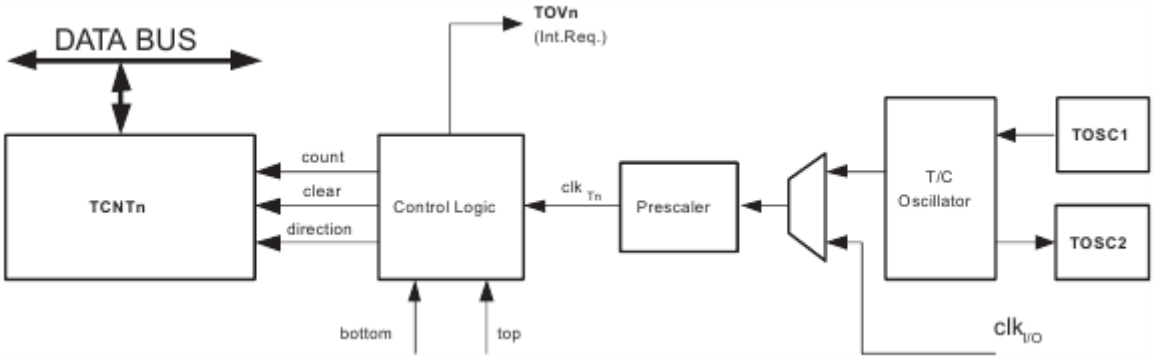


Почему кварц именно 32768Гц и почему его зовут часовым? Да все очень просто — 32768 является степенью двойки. Два в пятнадцатой степени. Поэтому пятнадцати разрядный счетчик, тикающий с частотой 32768 Гц, будет переполняться раз в секунду. Это дает возможность строить часы на обычной логической рассыпухе без каких либо заморочек. А в микроконтроллере AVR организовать часы с секундами можно почти без использования мозга, на рефлексах периферии.

Асинхронный режим таймера
Помните как работают таймеры? Тактовая частота с основного тактового генератора (RC внешняя или внутренняя, внешний кварц или внешний генератор) поступает на предделители, а с выхода предделителей уже щелкает значениями регистра TCNT. Либо сигнал на вход идет с счетного входа Tn и также щелкает регистром TCNT

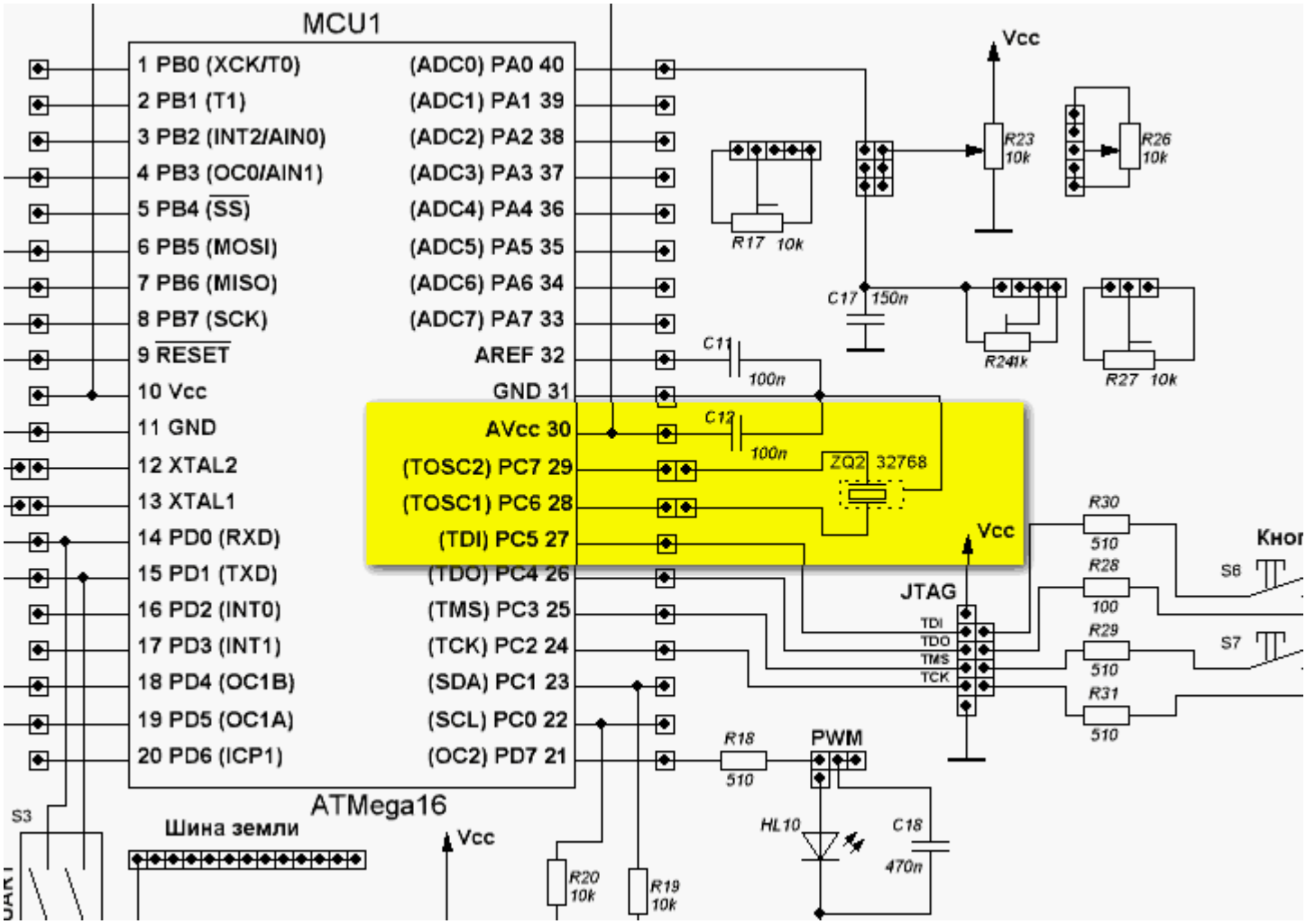


Структура же Timer/Counter2 немного отличается от остальных — у него нет счетного входа, зато есть возможность задействовать собственный тактовый генератор.



Для этого на выводы TOSC2 и TOSC1 вешается кварцевый резонатор. Низкочастотный, обычно это часовой кварц на 32768Гц. На Pinboard [2] он смонтирован справа от контроллера и подключается перемычками. Причем тактовая частота процессора должна быть выше как минимум в четыре раза. У нас тактовая от внутреннего генератора 8Мгц, так что нас это условие вообще не парит :)

Часовой кварц вешается просто на выводы. Без конденсаторов и каких либо заморочек.



И не нужно высчитывать количество тактов основного кварца, а если его нет, то заморачиваться на плавающую частоту встроенного RC генератора. Часовой кварц имеет куда более компактные размеры чем обычный кварц, да и стоит дешевле.

Также немаловажным является тот факт, что асинхронный таймер может тикать сам по себе, от часового кварца, ведь тактовая частота процессора ему не нужна, а это значит тактирование ядра контроллера (самое жручее, что

у него есть) можно отключить, загнав процессор в спячку, существенно снизив потребление энергии и просыпаясь только по переполнению таймера (1-2 раза в секунду), чтобы записать новые показания времени.

Конфигурирование
Для включения надо всего лишь установить бит AS2 регистра ASSR — и все, таймер работает в асинхронном режиме. Но есть тут одна фича которая мне стоила много головняков в свое время. Дело в том, что при работе от своего кварца все внутренние регистры таймера начинают синхронизироваться по своему же кварцу. А он медленный и основная программа может менять уже введенное значение гораздо быстрей чем оно обработается таймером.

Т.е., например, предустановил ты значение TCNT2, таймер на своей 32кгц молотилке его еще даже прожевать не успел, а твой алгоритм уже пробежал и снова туда что то записал — в результате в TCNT2 наверняка попадет мусор. Чтобы этого не случилось запись буфферизируется. Т.е. это ты думаешь, что записал данные в TCNT2, но на самом деле они попадают во временный регистр и в счетный попадут только через три такта медленного генератора.

Также буфферизируется регистры сравнения OCR2 и регистр конфигурации TCCR2

Как узнать данные уже внеслись в таймер или висят в промежуточных ячейках? Да очень просто — по флагам в регистре ASSR. Это биты TCN2UB, OCR2UB и TCR2UB — каждый отвечает за свой регистр. Когда мы, например, записываем значение в TCNT2 то TCNUB становится 1, а как только наше число из промежуточного регистра таки перешло в реальный счетный регистр TCNT2 и начало уже тикать, то этот флаг автоматом сбрасывается.

Таким образом, в асинхронном режиме, при записи в регистры TCNT2, OCR2 и TCCR2 сначала нужно проверять флаги TCN2UB, OCR2UB и TCR2UB и запись проводить только если они равны нулю. Иначе результат может быть непредсказуемым.

Да, еще один важный момент — при переключениях между синхронным и асинхронным режимом значение в счетном регистре TCNT может побиться. Так что для надежности переключаемся так:

- Запрещаем прерывания от этого таймера
- Переключаемся в нужный режим (синхронный или асинхронный)
- Заново настраиваем таймер как нам нужно. Т.е. выставляем предустановку TCNT2 если надо, заново настраиваем TCCR2
- Если переключаемся в асинхронный режим, то ждем пока все флаги TCN2UB, OCR2UB и TCR2UB будут сброшены. Т.е. настройки применились и готовы к работе.
- Сбрасываем флаги прерываний таймера/счетчика. Т.к. при всех этих пертурбациях они могут случайно установиться
- Разрешаем прерывания от этого таймера

Несоблюдение этой последовательности ведет к непредсказуемым и трудно обнаруживаемым глюкам.

Спящие режимы и асинхронный таймер
Т.к. асинхронный таймер часто используется в разных сберегающих режимах, то тут возникает одна особенность, раскладывающая целое поле из граблей.

Суть в том, что таймер, работающий от медленного кварца, не успевает за главным процессором, а в том дофига зависимостей от периферии — те же прерывания, например. И когда проц спит, то эти зависимости не могут реализоваться, в результате возникают глюки вроде неработающих прерываний или поврежденных значений в регистрах. Так что логика работы с асинхронным таймером и спящим режимом должна быть построена таким образом, чтобы между пробуждением и сваливаеним в спячку асинхронный таймер успел отработать несколько своих тактов и выполнил все свои дела.

Примеры:
Контроллер использует режим энергосбережения и отключения ядра, а пробуждается по прерываниям от асинхронного таймера. Тут надо учитывать тот факт, что если мы будем изменять значения регистров TCNT2, OCR2 и TCCR2, то уход в спячку нужно делат ТОЛЬКО после того, как флаги TCN2UB, OCR2UB и TCR2UB упадут. Иначе получится такая лажа — асинхронный таймер еще не успел забрать данные из промежуточных регистров (он же медленный, в сотни раз медленней ядра), а ядро уже отрубилось. И ладно бы конфигурация новая не применилась, это ерунда.

Хуже то, что на время модификаций регистров TCNT или OCR блокируется работа блока сравнения, а значит если ядро уснет раньше, то блок сравнения так и не запустится — некому его включить будет. И у нас пропадет прерывание по сравнению. Что черевато тем, что событие мы прошляпим и будем их терять до следующего пробуждения из спячки.
А если контроллер будится прерыванием по сравнению? То он уснет окончательно. Опаньки!
Вот и лови такой глюк потом.

Так что перед уходом в режимы энергосбережения надо обязательно дать асинхронному таймеру прожевать введенные значения (если они были введены) и дождаться обнуления флагов.

Еще один прикол с асинхронным режимом и энергосбережением заключается в том, что подсистема прерываний при выходе из спячки стартует за 1 такт медленного генератора. Так что даже если мы ничего не меняли, то обратно в спячку сваливаться нельзя — не проснемся, т.к. прерывания не успеют запуститься.

Так что выход из спячки и засыпание по прерыванию асинхронного таймера должно быть в таком виде:

- Проснулись
- …
- Что то сделали нужное
- …
- Заснули

И длительность операции между Проснулись и Заснули НЕ ДОЛЖНА БЫТЬ МЕНЬШЕ чем один тик асинхронного таймера. Иначе анабиоз будет вечным. Можешь delay поставить, а можешь сделать как даташит советует:

- Проснулись
- Что то сделали нужное
- Ради прикола записали что то в любой из буфферизиуемых регистров. Например, в TCNT было 1, а мы еще раз 1 записали. Ничего не изменилось, но произошла запись, поднялся флаг TCN2UB который продержится гарантированно три такта медленного генератора.
- Подождали пока флаг упадет
- Уснули.

Также не рекомендуется при выходе из спячки сразу же читать значения TCNT — можно считать лажу. Лучше подождать один тик асинхронного таймера. Или сделать прикол с записью в регистр и ожиданием пока флаг спадет, как было написано выше.

Ну и последний, но важный, момент — после подачи питания, или выхода из глубокой спячки, с отключением не только ядра, а вообще всей периферии, пользоваться медленным генератором настоятельно рекомендуется не раньше чем через **1 секунду** (не миллисекунду, а целая секунда!). Иначе генератор может еще быть нестабильным и в регистрах будет еще каша и мусор.

И, в завершение статьи, небольшой примерчик. Запуск асинхронного таймера на Atmega16 (Как полигон используется плата Pinboard [2])

Проект типовой, на базе диспетчера, одно лишь отличие — диспечтер переброшен на таймер0, чтобы освободить таймер2.

```
1  int main(void)
2  {
3  InitAll();                    // Инициализируем периферию
4  InitRTOS();                   // Инициализируем ядро
5  RunRTOS();                    // Старт ядра.
6
7  UDR = 'R';                    // Маркер старта, для отладки
8
9  SetTimerTask(InitASS_Timer,1000);      // Так как таймер в асинхронном режиме
10                                        // Выдержку для запуска инициализации таймера.
11
12 while(1)                      // Главный цикл диспетчера
13 {
```

```
14 wdt_reset();                    // Сброс собачьего таймера
15 TaskManager();                  // Вызов диспетчера
16 }
17 return 0;
18 }
```

Сама процедура инициализации таймера в асинхронном режиме сделана в виде конечного автомата. При первом запуске она взводит бит асинхронного режима и делает приготовления, после запускает сама себя снова же, через диспетчер, чтобы дать возможность еще чему либо проскочить в очереди, не блокируюя систему на ожидание.

При последующих входах проверяются флаговые биты готовности регистров таймера. Если они все по нулям, то мы на всякий случай зануляем флаги прерывания таймера, чтобы не было глюков и ложных срабатываний, а потом разрешаем нужное нам прерывание. И выходим.

```
   void InitASS_Timer(void)
   {
   if(ASSR & (1<<AS2))                                    //Если это второй вход то
           {
1          if (ASSR & (1<<TCN2UB | 1<<OCR2UB | TCR2UB) )         // проверяем есть ли хоть
2  один бит флаговый
3                  {
4                  SetTask(InitASS_Timer);                      // Если есть, то отправляем
5  на повторный цикл ожидания
6                  }
7          else                                                 // Если все чисто, то можно
8  запускать прерывания
9                  {
10                 TIFR  |= 1<<OCF2 | 1<< TOV2;           // Сбрасываем флаги прерываний, на
11 всякий случай.
12                 TIMSK |= 1<< TOIE2;                  // Разрешаем прерывание по
13 переполнению
14                 return;
15                 }
16         }
17
18 TIMSK &=~(1<<OCIE2 | 1<< TOIE2);       // Запрещаем прерывания таймера 2
19 ASSR  = 1<<AS2;                        // Включаем асинхронный режим
20 TCNT2 = 0;
21 TCCR2 = 5<<CS20;                       // Предделитель на 128 на 32768 даст 256 тиков в
22 секунду
23                                        // Что даст 1 прерывание по переполнению в
   секунду.
   SetTask(InitASS_Timer);                        // Прогоняем через диспетчер, чтобы зайти
   снова.
   }
```

А дальше движуха идет уже по прерыванию. Раз в секунду у нас переполняется регистр асинхронного таймера и генерит прерывание. Тут уже делаем что хотим. Я просто переменную увеличивал и выводил в UART порт.

```
1 ISR(TIMER2_OVF_vect)                  // Прерырвание по переполнению таймера 2
2 {
3 UDR = i;
4 i++;
5 }
```

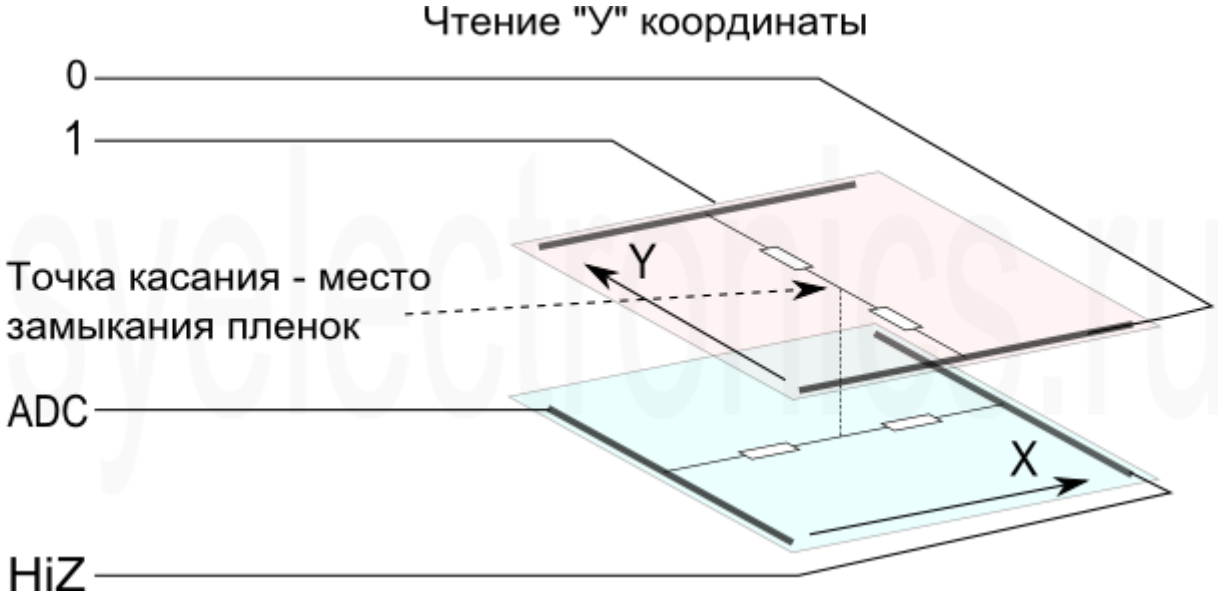
Можно было сделать переменные содержащие часы:минуты:секунды и щелкать этими переменными со всей их логикой переполнения часов/минут, но мне было лень. И так все понятно.

- Архив с проектом под AvrStudio + WinAVR для этой статьи [3]

Работа с резистивным сенсорным экраном

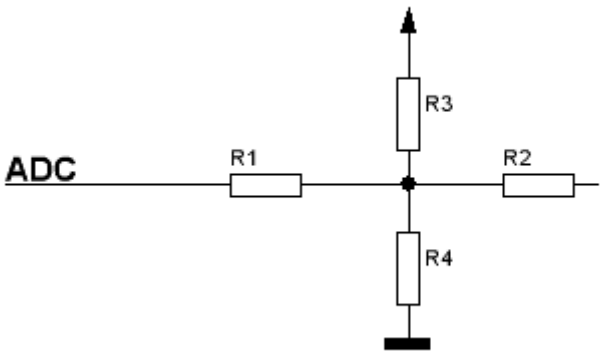
Хоть резистивный touchscreen и является устаревшим и активно вытесняется емкостными сенсорами, но тем не менее он еще не скоро канет в Лету. Во первых из-за простоты и дешевизны, а во вторых из-за элементарной работы с ним.

Конструктив
Итак, как он устроен. Там все очень и очень просто. Есть две пленки, сделанные из проводящего материала, а между ними гранулы диэлектрика. Когда касаемся пальцем, то продавливаем зазор между пленками и контачим верхнюю на нижнюю. Ну, а определить координаты касания уже дело несложное.



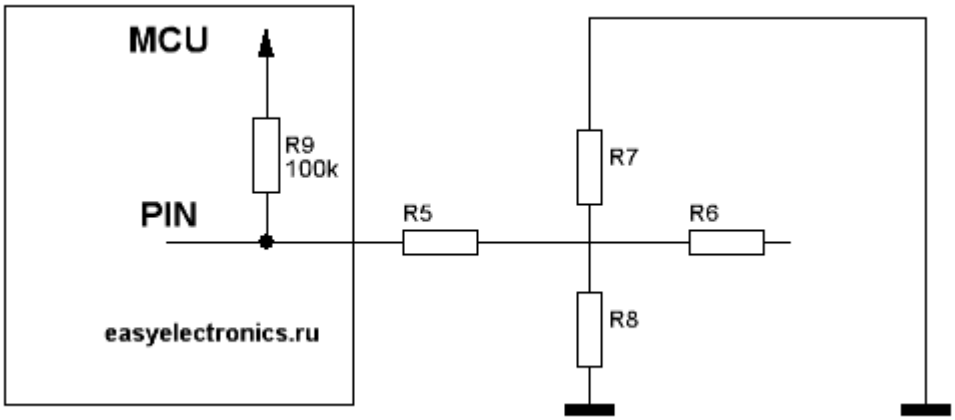
Для этого на каждую пленку нанесено по два электрода. Слева-справа на одной и сверху-снизу на другой. Крест на крест, в общем. Поскольку сопротивление пленки довольно большое, под сотни ом, то образуются как бы два перпендикулярных резистора, висящие друг над другом.

В точке касания они замыкаются между собой и вуаля, получается такая схема:



Теперь достаточно концы одной пленки подтащить к шинам питания (банально заведя туда лог 0 и лог 1), чтобы она образовала обычный резистивный делитель. А с другой пленки снять получившееся напряжение. Которое будет пропорционально координате. Так как сопротивления входа АЦП огромно, то жалкие сотни ом, что составляет остаток пленки (R1) нам никак не помешает. Также не помешает и болтающийся конец (R2).
Теперь меняем положение, растягивая между шинами питания уже другую пленку и снимаем вторую координату. Элементарно!

Вилы в стогу
Но при попытке взять схему в лоб, то получаем первые вилы в бок. Дело в том, что если мы будем тупо менять по очереди пленки и сканировать значения, то много мы не намеряем. Т.к. когда касания нет, то пленки не перескаются, а значит каналы АЦП во время измерения висят в воздухе, ловя всякий мусор. И как их отличить от нажатия? Да никак! АЦП без разницы, что на вход пришло.
Так что нам надо замер координат делать не непрерывно, а тогда и только тогда, когда есть касание. До этого на тачскрин даже не отвлекаться.
Делается это просто — мы одну пленку подтягиваем к земле. А вторую вешаем на вход с pullup. Был тачскрин, стала обычная кнопка.



Ее мы лениво опрашиваем, в ожидании нажатия. Но тут прячутся вторые вилы, на которые мы будем нарываться еще не раз и не два. Дело в том, что тачскрин это две проводящие пленки. Большие плоские пленки, находящиеся очень близко друг от друга. Ничего не напоминает? Правильно — конденсатор, мать его. А значит у него нефиговая емкость. И если ты его переключишь в режим отслеживания нажатия и тут же проверишь — получишь нажатие. Т.к. емкость еще не успела зарядиться, а значит коротит. Так что после перевода порта в режим опроса надо подождать около миллисекунды. А только потом начинать щупать. То же касается, кстати, и режима смены замеров координат. Начнешь быстро переключать пленки с координаты на координату — получишь полную херню на выходе. Переключил, подождал, замерил. Переключил, подождал, замерил. Только так.

Под это дело я набросал небольшой пример кода для ATMega16 на Pinboard [1].

Код
Код в виде обычного конечного автомата. Правда он разнесен на процедуру и прерывание.

```
1   //=======================================
2   //Область задач
3   //=======================================
4   void ScanTouch(void)
5   {
6   switch(TouchState)
7           {
8           case 0: // Ожидание нажатия. Подготовка.
9                   {
10                  DDRA  &=0xF0;
11                  PORTA &=0xF0;
12
13                  DDRA  |=1<<XA | 1<<XB;          // XA и XB гонят землю
14                  PORTA |=1<<YA | 1<<YB;          // YA и YB слушают уровень на входе с
15  подтяжкой
16
17                  TouchState = 1;                 // Переход к стадии  проверки.
18
19                  SetTimerTask(ScanTouch,10);     // Выдержка для установления состояния и
20  зарядки емкости.
21
```

```
22                  break;
23                  }
24          case 1: // Проверка нет ли нажатия
25                  {
26                  if(PINA & 1<<YA)                        // Если на входе 1, т.е. нажатия
27  нет
28                          {
29                          TouchState = 0;                 // Возврат на прошлый шаг.
30                          SetTimerTask(ScanTouch,10);
31                          }
32                  else                                    // Если 0, т.е. нажатие. Готовим
33  замер!
34                          {
35                          PORTA &=0xF0;                   // Зануляем четыре бита, чтобы с
36  чистого листа.
37                          DDRA &=0xF0;
38                                                          // Сейчас Y пленка гонит
39  потенциал, X считывает.
40                          PORTA |=1<<YA;                  // YA на Vcc  YB на GND остальные
41  в HiZ нюхают АЦП
42                          DDRA |=1<<YA|1<<YB;             // -
43
44                          TouchState=2;                   // Переход на другую стадию.
45                          SetTimerTask(ScanTouch,10);     // Выдержка на переходный процесс.
46                          }
47                  break;
48                  }
49
50          case 2: // Запускаем АЦП для снятия координат
51                  {
52                  TouchState = 3;
53                  ADMUX = 1<<REFS0 | 1<<ADLAR | 2<<MUX0;                  //Меняем канал АЦП
54  на чтение X
55                  ADCSRA = 1<<ADEN | 1<<ADIE | 1<<ADSC | 3<<ADPS0;       // Запускаем
56  преобразование.
57                  break;                                                  // До встречи в
58  обработчике прерывания.
59                  }
60
61          case 5: // Координаты сняты. Можно обрабатывать.
62                  {
63                  TouchState = 0;
64                  if (!UDR_Busy)                  // Если UART не занят
65                          {
66                          UDR_Busy = 1;           // Занимаем его
67
68                          OutBuff[0] = 'Y';       // Пишем в буфер мессагу
69                          OutBuff[1] = '=';
70
71                          BIN8toASCII3(OutBuff[2],OutBuff[3],OutBuff[4],coord_Y);       //
72  Вписываем координаты
73
74                          OutBuff[5] = ' ';
75                          OutBuff[6] = 'X';
76                          OutBuff[7] = '=';
77
78                          BIN8toASCII3(OutBuff[8],OutBuff[9],OutBuff[10],coord_X);
79
80                          OutBuff[11] = 10;               //CR - не забываем перевод каретки
81                          Buff_index = 0;                 // Обнуляем индекс
82                          UDR = OutBuff[Buff_index];      // Поехали!
83                          }
84
```

```
85                          SetTimerTask(ScanTouch,100);   // Повтор сканирования. Не
86  торопимся
87                          break;
88                          }
89          default:
90                  {
91                  TouchState = 0;
92                  }
93          }
94  }
95
96  // Продолжаем веселиться в обработчике прерывания от АЦП
97  ISR(ADC_vect)
98  {
99  PORTA &=0xF0;           // Зануляем четыре бита, чтобы с чистого листа.
100 DDRA &=0xF0;
101
102 if(TouchState == 3)
103         {
104         coord_X = ADCH;         // Прошлый замер дал X
105
106         // Сейчас уже X пленка гонит потенциал, Y считывает
107         PORTA |=1<<XA;          // XA на Vcc  XB на GND остальные в HiZ нюхают АЦП
108         DDRA |=1<<XA|1<<XB;             // -
109
110         ADMUX = 1<<REFS0 | 1<<ADLAR | 3<<MUX0;         // Перебрасываем канал АЦП на
    другую пленку
            TouchState = 4;                                 // Перекидываем автомат

            ADCSRA = 1<<ADEN | 1<<ADIE | 1<<ADSC | 3<<ADPS0;       // Запускаем конверсию и
    до встречи тут же.
            }
    else if (TouchState == 4)               // Второй вход в прерывание
            {
            coord_Y = ADCH;                 // Прошлый замер дал Y

            TouchState = 5;                 // Стадия обработки
            SetTask(ScanTouch);             // Обе координаты считаны. Готово. Вызваем задачу
    готовности.
            }
    }
```

Выброс данных в UART идет следующим образом:

```
1  ISR(USART_TXC_vect)
2  {
3  Buff_index++;           // Увеличили индекс
4  if(Buff_index != 12)    // Если мессадж не ушел
5          {
6          UDR = OutBuff[Buff_index];      // Шлем дальше
7          }
8  else
9          {
10         Buff_index = 0;                 // Иначе все зануляем.
11         UDR_Busy = 0;
12         }
13 }
```

Все очень и очень просто, без задержек и тормозов. Все на конечных автоматах и диспетчере. Но диспетчер тут не принципиален, может быть что угодно. Хоть тот же флаговый автомат. Главное, чтобы была служба таймеров. Весь проект в архиве в конце статьи.

BIN8toASCII3 это макрос переводящий бинарное значение в три разряда ASCII символов, пригодных к выводу в терминалку.

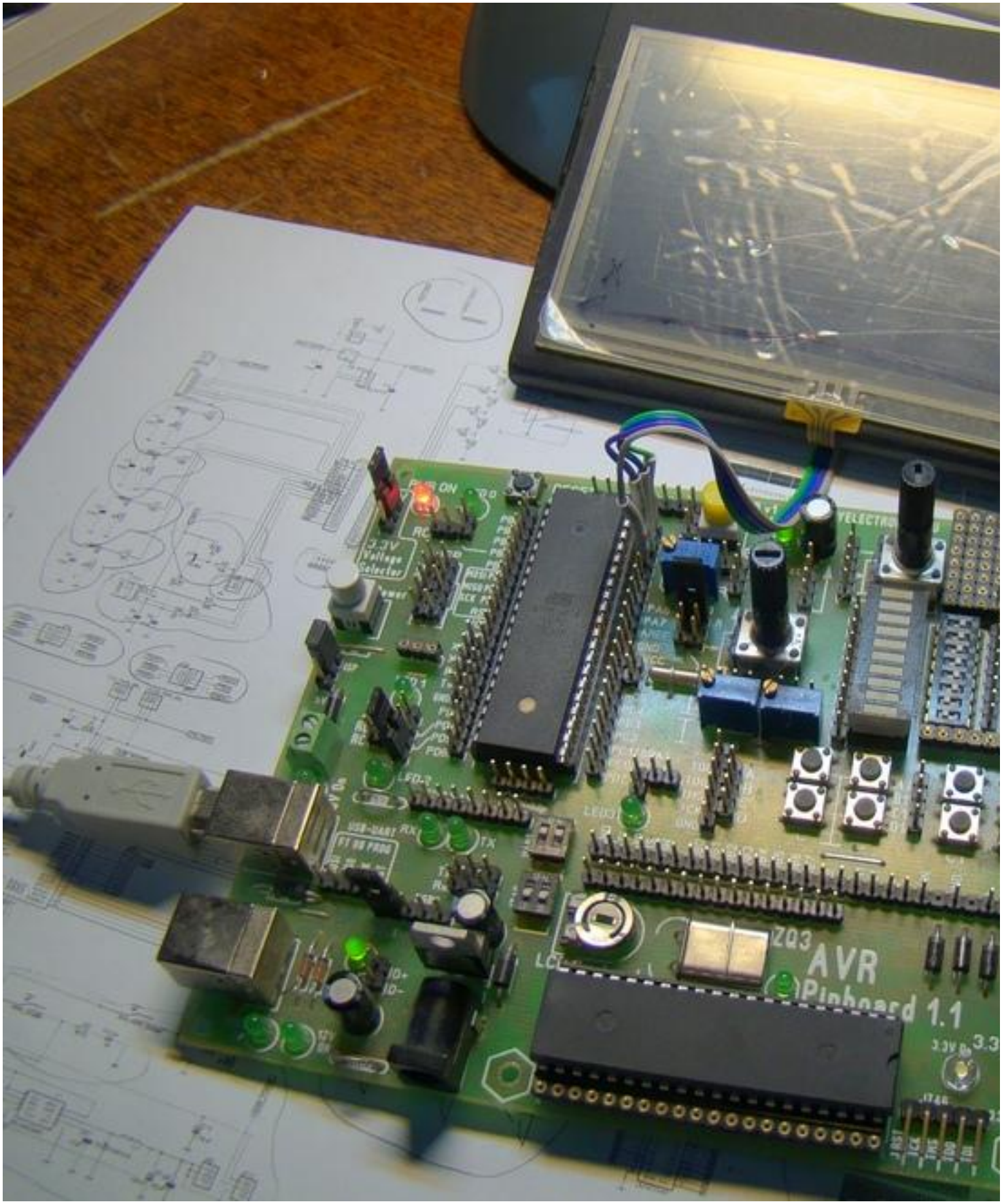
Выглядит оно так:

```
1  // Bin to ASCII
2  // IN = value
3  // A B C = ascii code of digits
4  #define BIN8toASCII3(a_,b_,c_,in_)                                          \
5  do                                                                          \
6  {                                                                           \
7  asm volatile(                                                               \
8                  "LDI %A[RA],-1+'0'"                          "\n\t"         \
9  "bcd1%=:"       "INC %A[RA]"                                 "\n\t"         \
10                 "SUBI %A[RC],100"                            "\n\t"         \
11                 "BRCC bcd1%="                                "\n\t"         \
12                 "LDI %A[RB],10+'0'"                          "\n\t"         \
13 "bcd2%=:"       "DEC %A[RB]"                                 "\n\t"         \
14                 "SUBI %A[RC],-10"                            "\n\t"         \
15                 "BRCS bcd2%="                                "\n\t"         \
16                 "SBCI %A[RC],-'0'"                           "\n\t"         \
17                                                                             \
18                  : [RA]"=a" (a_), [RB]"=a" (b_), [RC]"=a" (c_)                  \
19                  : "[RC]" (in_)                                             \
20          );                                                                 \
21                                                                             \
22 }                                                                           \
23 while(0)
```

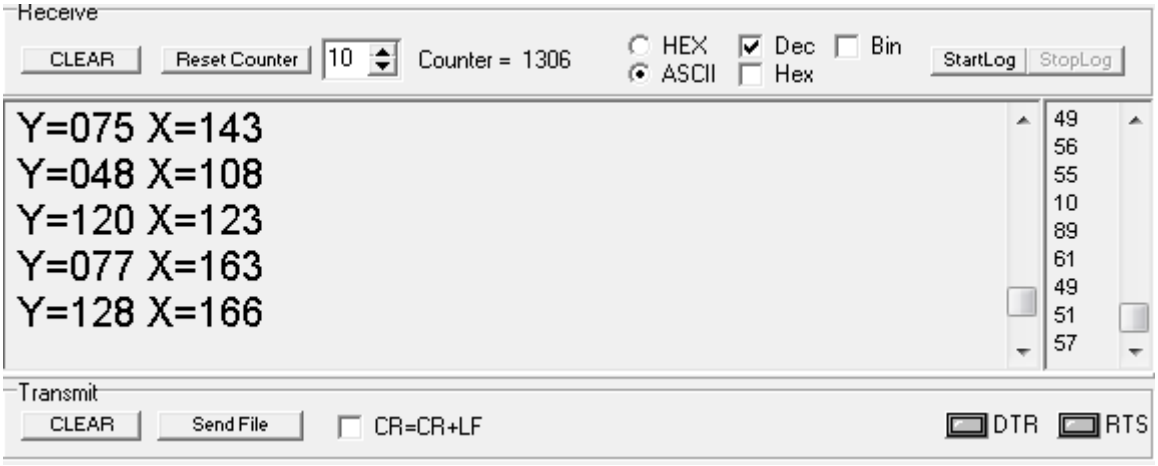
Нет, это не УЖОСНАХ, а GCCшная ассемблерная вставка. Я корячил алгоритм BIN2ASCII и так и эдак, потом решил, что красивей, короче и быстрей чем на ассемблере не получится и воткнул его в код так. О том как писать такие вставки в GCC неплохо написал **Alatar** в нашем сообществе [2]. Если кто не читал, то рекомендую. Также, если будут желающие, то я дополнительно постараюсь разжевать эту тему до состояния манной каши.

В железе
Сам touchscreen надыбал в Китае, на Dealextrem. Обошелся он мне в пару баксов. Можно еще не очень дорого купить у всяких ремонтников сотовых. Конкретно этот от Nintendo DS, но там множество вариантов с разной диагональю и по различной цене.

Аккуратно припаялся к пленочке, выставив минимальную температуру паяльника (около 230 градусов), чтобы не поплавить нежную пленочку. А потом, для демонстрации, прибил все к какому-то блокнотику, что завалялся под рукой.



Да подключил все лапшой к контроллеру. Данные традиционно вывожу на терминалку.



- **Проект в AVR Studio 4 для ATMega16** [3]

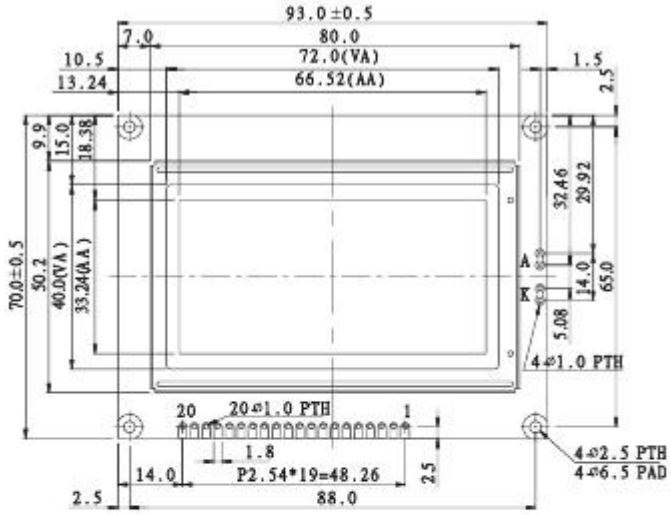
Работа с графическим дисплеем WG12864 на базе контроллера KS0107

Обычно для вывода информации сигнального дисплея на HD44780 более чем достаточно. Но иногда нужно нарисовать картинку, график или хочется сделать красиво, с модными менюшками. Тут на помощь приходят графические дисплеи. Одним из самых простых и доступных является дисплей на контроллере KS0107 или аналоге. Например, WG12864A от Winstar. Сам дисплей вполне свободно достается, имеет довольно большой размер (диагональ около 80мм) и разрешение 128x64 пикселя. Монохромный. Цена вопроса 400-500р.

Вот такой вот:

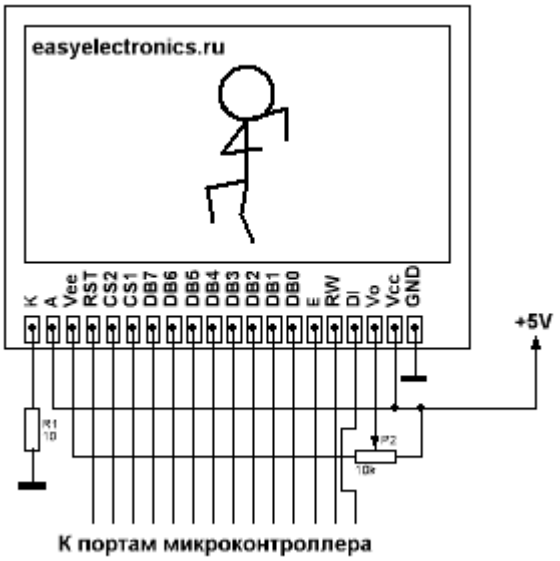
Подключение
Управление дисплеем параллельное. Есть шина данных и линии задания направления и вида данных. Это, кстати, один из минусов — требует очень много проводов управления. Занимает почти 16 линий. Но зато и очень прост в работе.

Итак, если взять тот, что у меня WG12864A-TGH-VNW то у него следующая распиновка:



Pin NO.	Symbol	Function
1	Vss	GND
2	Vdd	Power supply (+5V)
3	Vo	Contrast Adjustment
4	D/I	Data/instruction
5	R/$\overline{W}$	Data read/write
6	E	H→L Enable signal
7	DB0	Data bus line
8	DB1	Data bus line
9	DB2	Data bus line
10	DB3	Data bus line
11	DB4	Data bus line
12	DB5	Data bus line
13	DB6	Data bus line
14	DB7	Data bus line
15	CS1	Chip select for IC1
16	CS2	Chip select for IC2
17	$\overline{RST}$	Reset
18	Vee	Negative voltage output
19	A	Power supply for LED+(4.2V) RA=0Ω
20	K	Power supply for LED (0V)

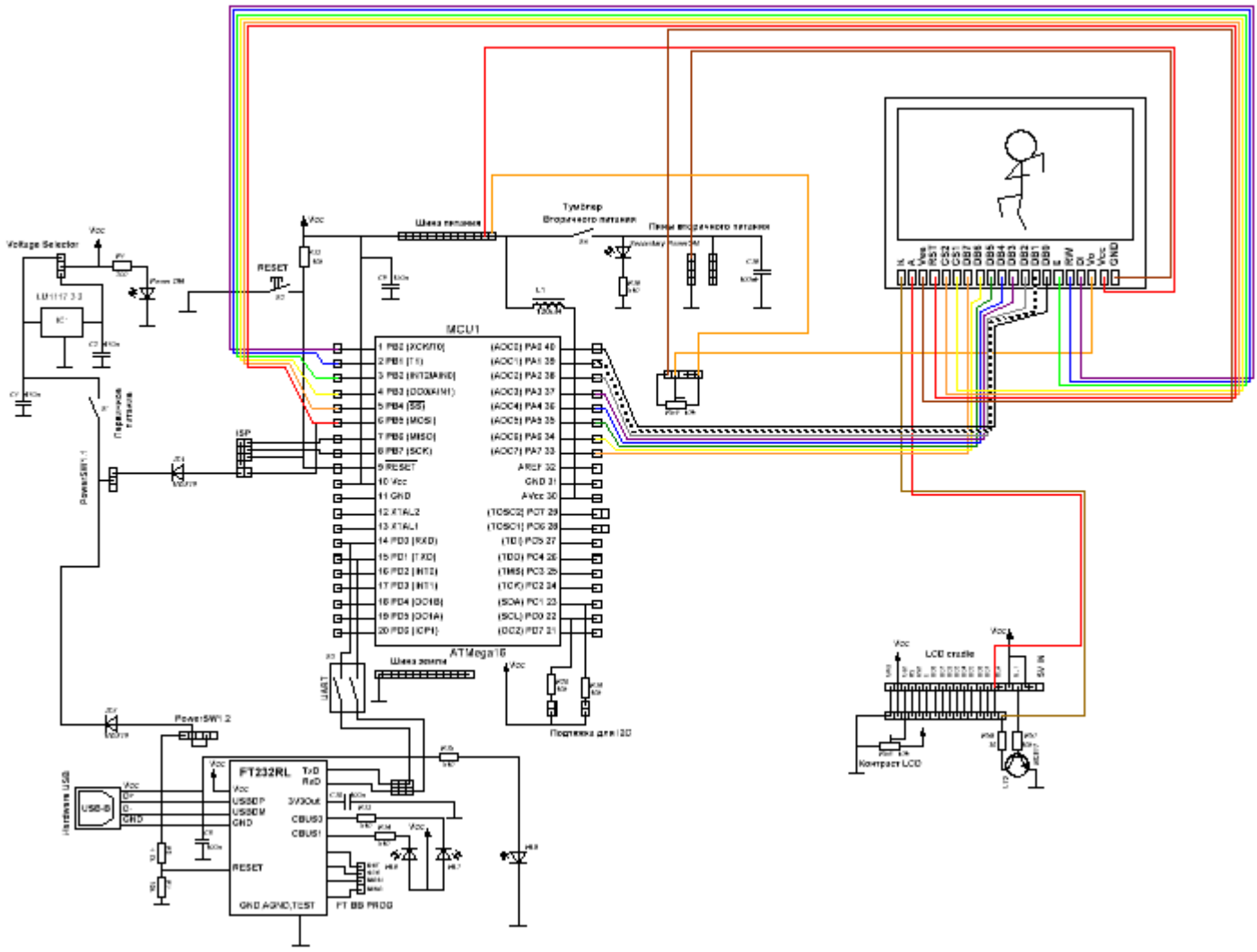
- **Vdd и Vss** это питание, оно у него пятивольтовое.
- **Vee** — источник отрицательного напряжения. Там примерно минус 5 вольт. Есть не на всех моделях этих дисплеев, у Winstar о наличии такой фенечки говорит буква V в маркировке. WG12864A-TGH-**V**NW
- **Vo** — напряжение регулировки контраста. Туда подается либо 0…5 вольт, либо от -5 до 5, в зависимости от модели и температурного диапазона. Обычно более морозостойкие дисплеи требуют отрицательное напряжение. Схема включения простая:



- **D/I** — Данные/команда. Логический уровень на этом выводе определяет предназначение кода на шине данных. 1 — данные, 0 — команда.
- **R/W** — Чтение/Запись. Уровень на этой ноге задает тип действия. 1 чтение, 0 запись.
- **E** — Строб, синхронизирующий импульс. Дрыг этой вожжи вверх-вниз проворачивает шестеренки в интерфейсе контроллера.

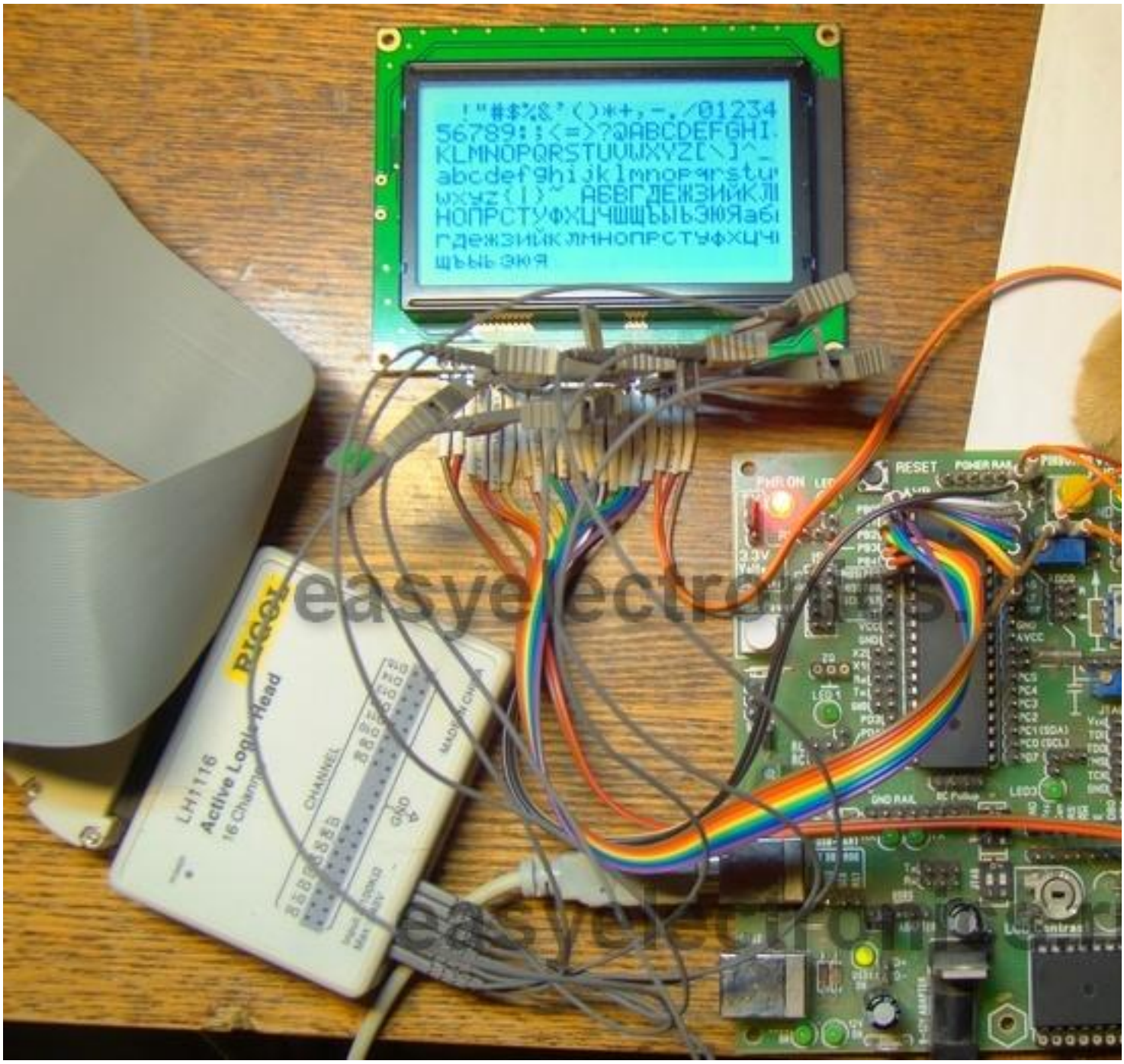
- **DB0..7** — Шина данных, на которую мы выдаем нужный нам код.
- **CS1 и CS2** — выбор контроллера.
- **RST** — сигнал сброса. Ноль на этой линии сбрасывает контроллеры в ноль. Но не сбрасывает видеопамять, только текущую адресацию.
- **A и K** — питание светодиодной подсветки. Там, как правило, обычные светодиоды, так что напрямую туда 5 вольт подавать нельзя. Только через ограничительный резистор. Ток потребления подсветки весьма велик, около 200мА, падение напряжения в районе 4 вольт. На пяти вольтовом питании ограничетельный резистор должен быть порядка 5-10 Ом.

К контроллеру (ATMega16 на Pinboard [1]) я подключил все следующим образом.



Крупным планом

Данные полностью легли на PORTA, а управление на PORTB. В качестве резистора подстройки контраста я взял многооборотный переменник, что так кстати стоит рядом для подобных случаев. Питание подсветки взял с колодки от дисплеяя. Благо там все уже готово, даже управление от транзистора есть :) Правда я ее просто включил.

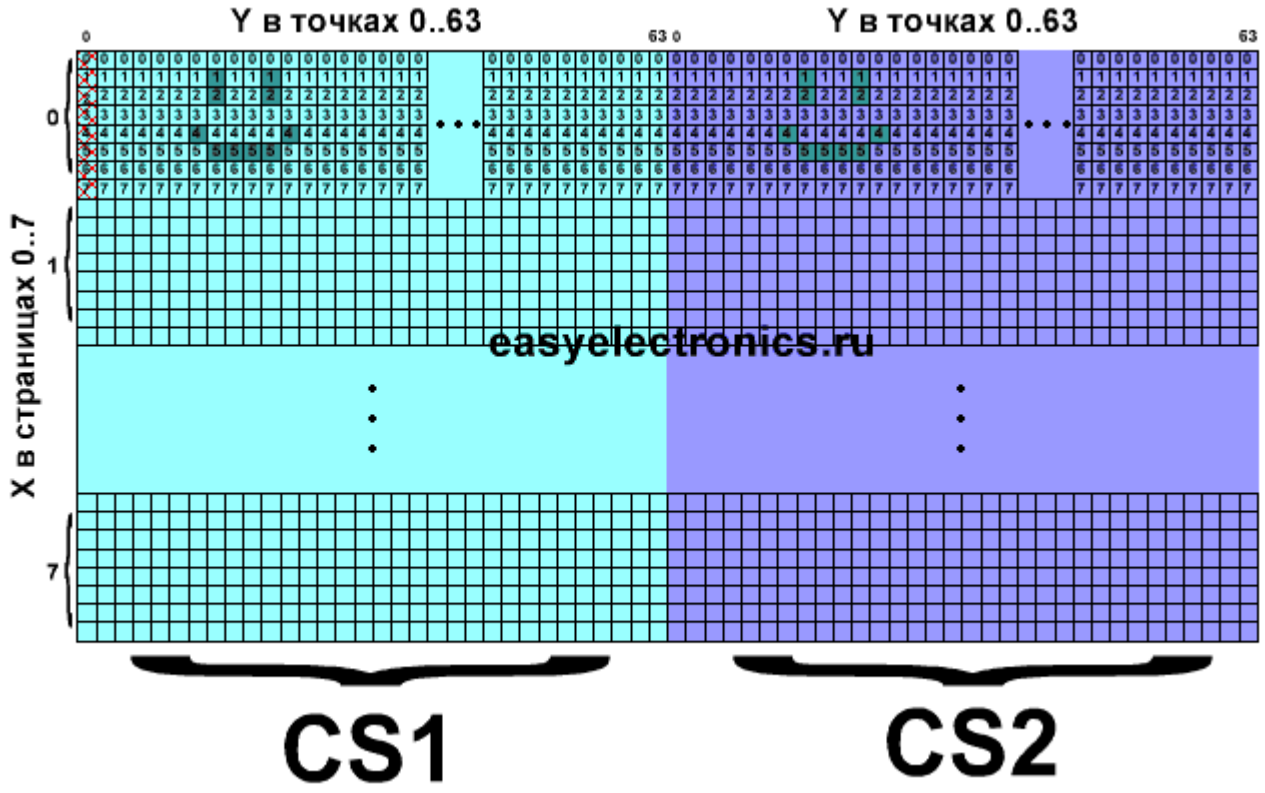


Двое из ларца, одинаковых с лица. Адресация

Контроллер CS0107 он может организовать матрицу только 64x64. А у нас в дисплее вдвое большая 128x64. Значит стоят два контроллера. Один отвечает за правую половину экрана, другой за левую.

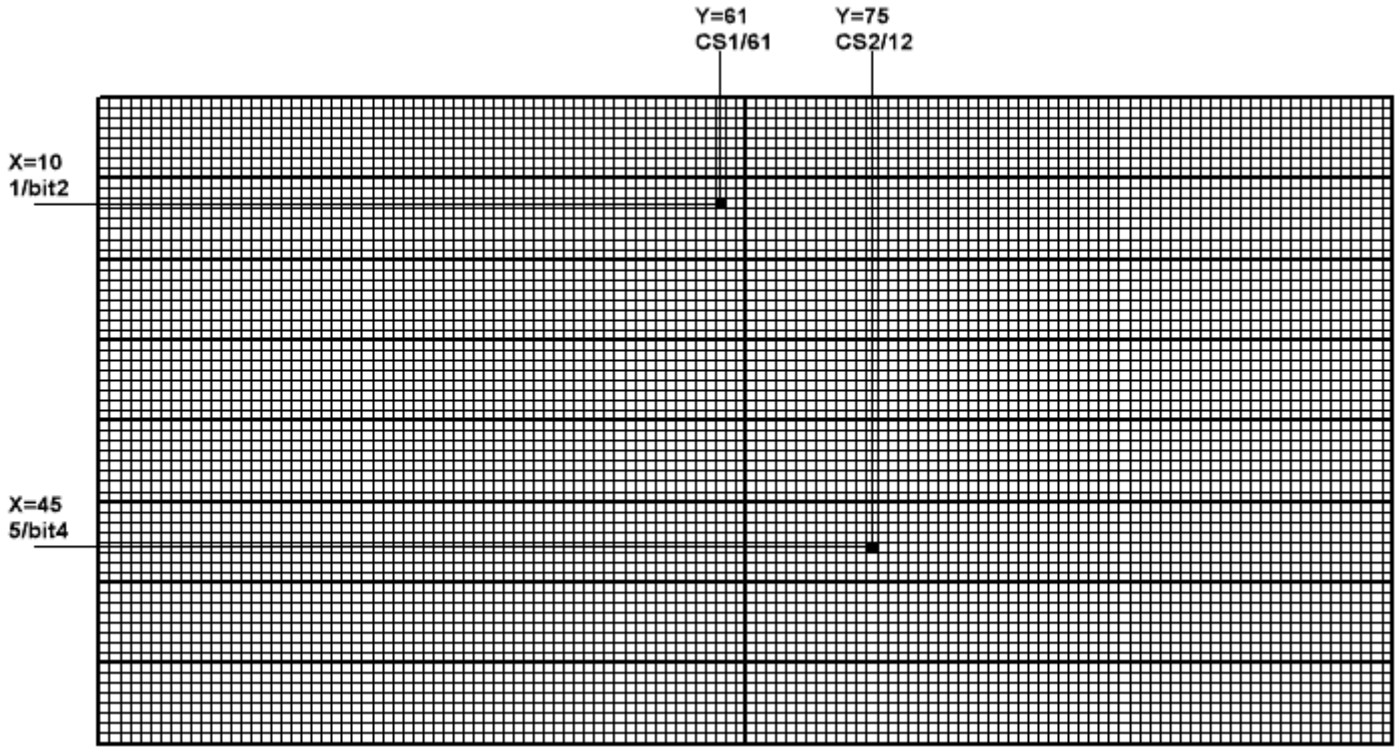
Он представляет собой этакую микросхему памяти, где все введенные данные отображаются на дисплее. Каждый бит это точка. Кстати, для отладки удобно юзать, выгружая туда разные данные, а потом разглядывая этот дамп).

Карта дисплея выглядит так:



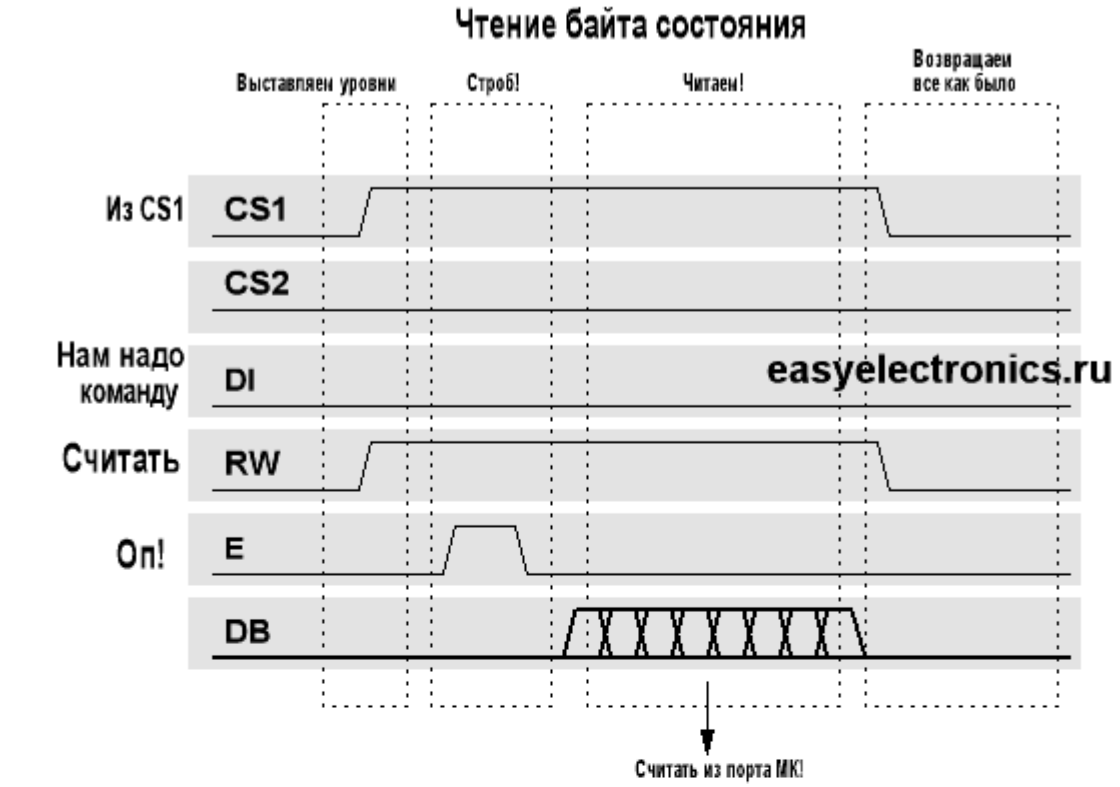
Байты укладываются в два контроллера страницами по 64 байта. Всего 8 страниц на контроллер.

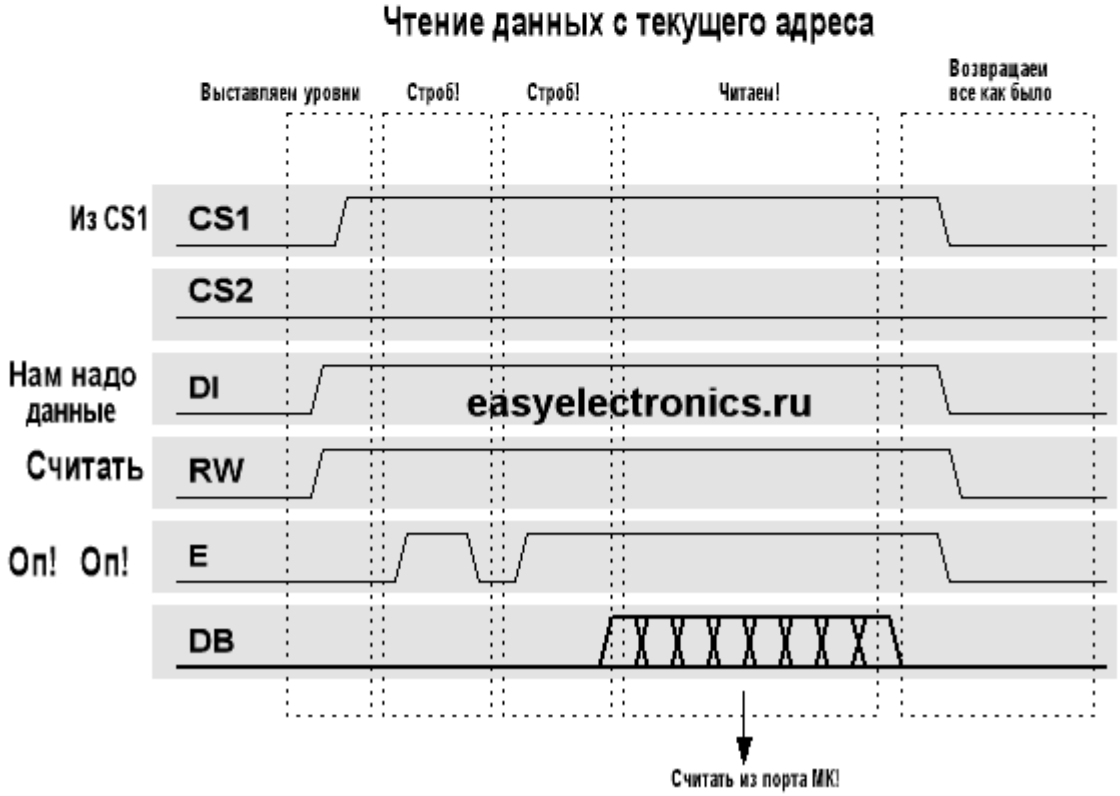
Так что для того, чтобы выставить точку с координатами на экране, например, X = 10, Y=61 надо вычислить в каком контроллере она находится. Первый до 63, второй после, если адрес во втором контроллере, то надо вычесть 64 из координаты. Затем вычислить страницу и номер бита. Страница это X/8, а номер бита остаток от деления (X%8). Потом нам надо считать нужный байт из этой страницы (если мы не хотим затронуть остальные точки), выставить в нем наш бит и вернуть байт на место.



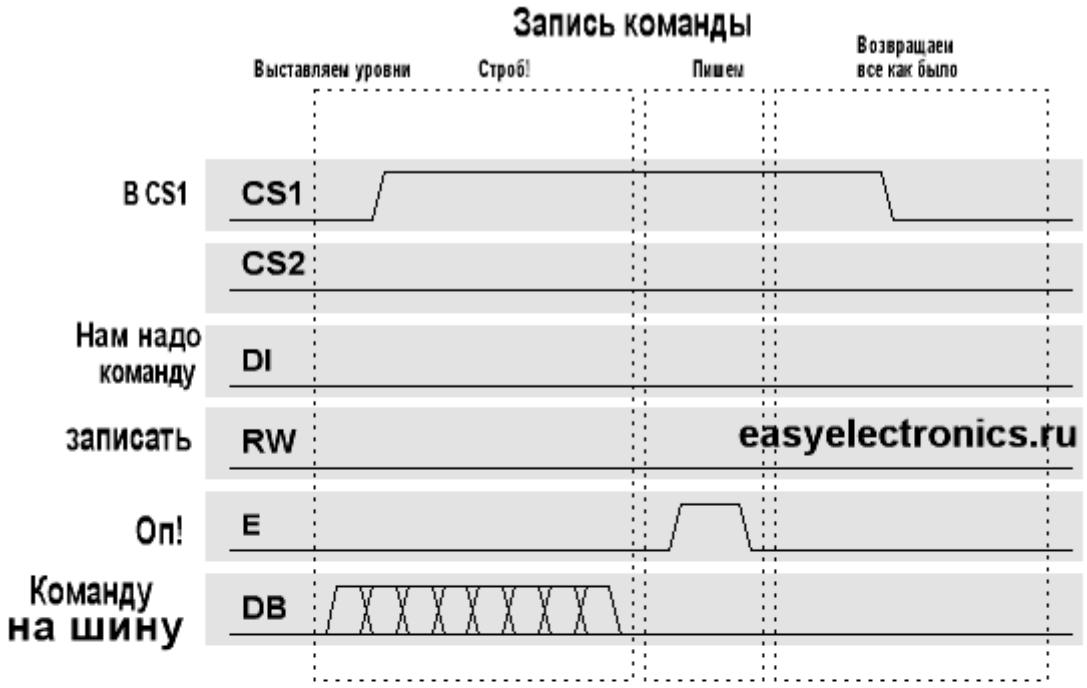
Протокол обмена
Тут все просто, без каких либо изысков. Выставляем на линиях RW, DI что мы хотим сделать, линиями CS1 и CS2 выставляем к кому обращаемся. На шину данных выдаем нужное число и поднимаем-опускаем линию строба. Опа! Впрочем, есть одна тонкость. Для чтения данных строб нужно дернуть дважды, т.к. предварительно данные должны попасть в регистр-защелку. Для чтения же флага состояния такой изврат не нужен. Вот примеры временных диаграм для разных режимов.

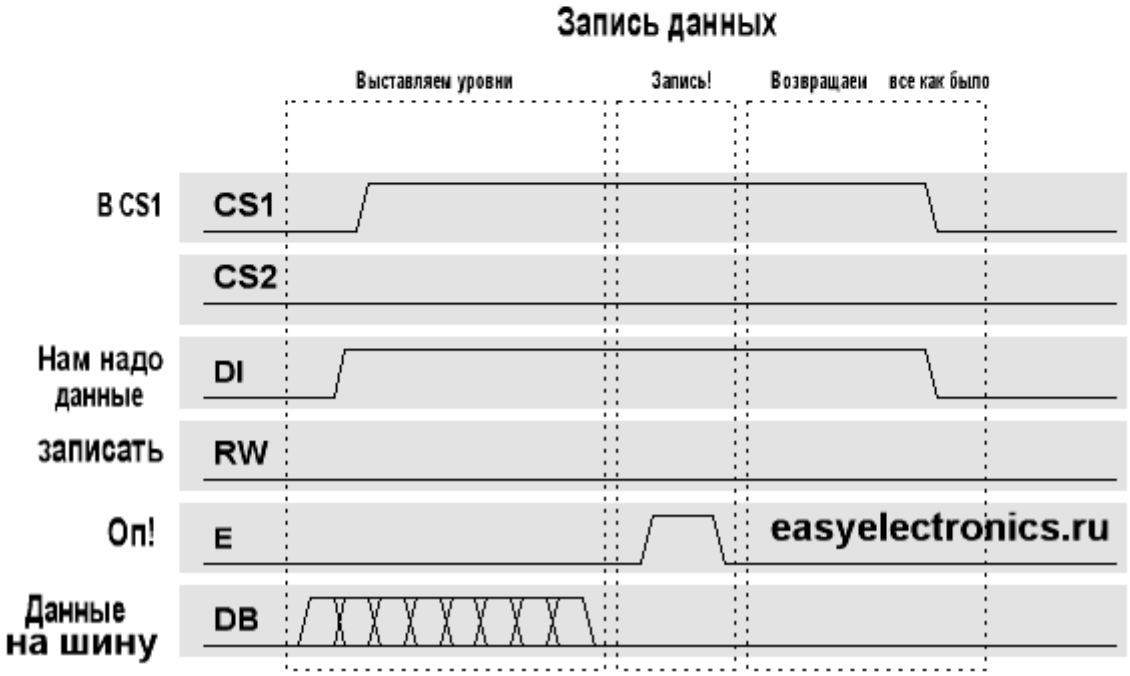
Чтение





И запись. Причем запись, в отличии от чтения, можно делать сразу в оба контроллера. Конечно одновременно писать данные в контроллер смысла имеет мало, разве что захочишь двойную картику получить :) А вот команды обычно пишут срзау в оба.

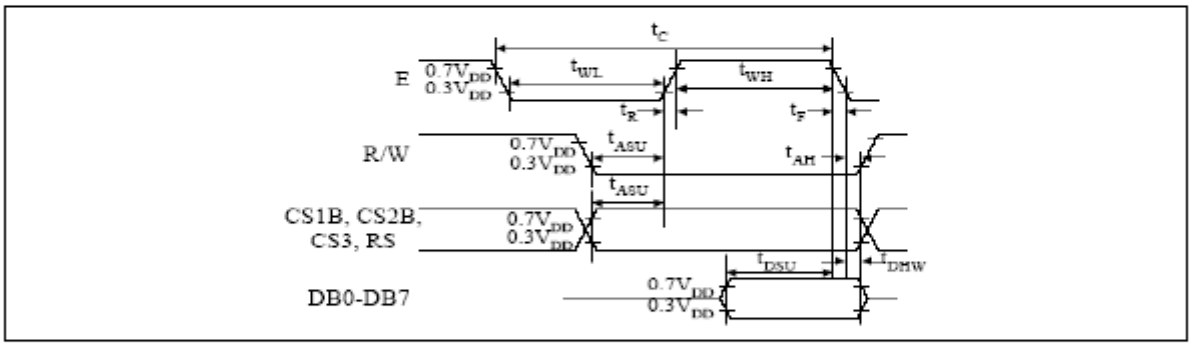




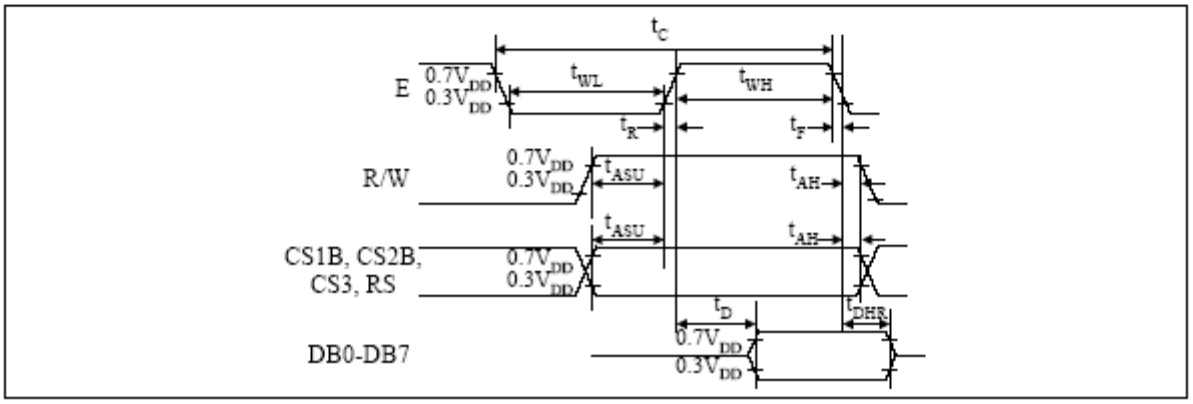
Кстати, еще есть одна особенность — выводы CS **могут быть инверсными.** Это зависит от модели дисплея. Так что это надо учитывать.

Временные диаграммы, т.е. сдвиг фронтов между собой по времени может быть разным у разных контроллеров. Где то быстрей, где то медленней. Но в целом 1мкс обычно хватает. В реале обычно меньше. Лучше поглядеть в даташите на конкретный контроллер (не дисплей, в ДШ на дисплей обычно редко есть описание самого контроллера). Там обычно есть таблица вида:

Characteristic	Symbol	Min	Typ	Max	Unit
E cycle	tcyc	1000	-	-	ns
E high level width	twhE	450	-	-	ns
E low level width	twlE	450	-	-	ns
E rise time	tr	-	-	25	ns
E tall time	tf	-	-	25	ns
Address set-up time	tas	140	-	-	ns
Address hold time	tah	10	-	-	ns
Data set-up time	tdsw	200	-	-	ns
Data delay time	tddr	-	-	320	ns
Data hold time (write)	tdhw	10	-	-	ns
Data hold time (read)	tdhr	20	-	-	ns



MPU Write Timing



MPU Read Timing

Где указаны максимально допустимые временные интервалы. Если они будут меньше или больше, то дисплей скорей всего не будет работать. Или будет работать с ошибками. Если у вас на дисплей в процедуре заливки или очистки экрана полезли всякие левые пиксели и прочие артефакты — значит тайминги не выполняются. Также рекомендую проверять тайминги на чтение и на запись. Делается это просто — гоним последовательно сначала чтение, а потом запись обратно. Если ничего не изменилось — значит все ок. Появились искажения? Крутите

временные задержки. Только рекомендую читать не пустоту вида 0x00, а что нибудь более веселое залить, например, шахматную доску из пикселей. По очереди 0x55 и 0xAA.

Система команд.
Она тут простейшая.

Команда	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Назначение
Отображение ВКЛ/ВЫКЛ	L	L	L	L	H	H	H	H	H	L/H	Управляет вкл/выкл отображения. Не влияет на внутреннее состояние и данные ОЗУ изображения. L: ВЫКЛ H: ВКЛ
Установить Адрес	L	L	L	H	Адрес Y (0 ~ 63)						Заносит адрес Y в счётчик адреса Y
Установить Страницу (адрес X)	L	L	H	L	H	H	H	Страница (0 ~ 7)			Заносит адрес X в регистр адреса X
Начальная Строка Отображения	L	L	H	H	Начальная строка отображения (0 ~ 63)						Скроллинг вверх. На сколько пикселей сдвинуть адресное пространство. При этом уехавшее вверх, за экран, вылезет снизу, словно мы провернули экранную область как барабан
Чтение Состояния	L	H	BUSY	L	ON/OFF	RESET	L	L	L	L	Чтение состояния. BUSY L: Готовность H: Выполняется команда ON/OFF L: Отображение ВКЛ H: Отображение ВЫКЛ RESET L: Нормальный режим H: Сброс
Запись Данных Изображения	H	L	Данные для записи								Записывает данные (DB0:7) в ОЗУ данных изображения. После записи инструкции, адрес Y увеличивается на 1 автоматически.
Чтение Данных Изображения	H	H	Данные для чтения								Читает данные (DB0:7) из ОЗУ данных изображения на шину данных. После чтения адрес Y сам увеличивается на 1 автоматически

Инициализация дисплея элементарная, в отличии от HD44780, где надо переключать режимы, включать-выключать разные курсоры и отображения.

Надо после сброса задать начальные координаты (адрес точки и страница), значение скролинга (обычно 0) и включить отображение. При включении на дисплее может быть мусор. Поэтому отображение обычно включают после очистки видеопамяти.

У меня ициализация, по командам, выглядит так:

- 0x3F — включить
- 0×40 — Адрес Y=0
- 0xB8 — Страница X=0
- 0xC0 — Скролл = 0 (т.е. с самого верха)

Ну и потом еще заливка сразу в оба контроллера.

Код
Итак, приступим к коду. Чтобы было наглядней я все операции с ногами расписал в виде макросов. Заодно будет гораздо проще все перенести на другую архитектуру. Просто написав другие макросы, не правя сам код :)
Весь код можно поглядеть в нашей кодосвалке:
lcd_wg128.h [2]
lcd_wg128.c [3]

Покажу тут лишь характерные моменты работы с дисплеем.

Записью команд и данных занимаются следующие функции:

```
1  // Пишем команду в выбранный контроллер
2  void LCD_WR_COM(u08 cmd,u08 CSC)
3  {                        // Поднятие сигналов:
4  LCD_SET_CMD;             // Команда
5  LCD_SET_W;               // Запись
6
7  ON_CS(CSC);              // Выбор чипа
8
9  LCD_DATA_INS(cmd);       // Команду на шину данных
10
11 NOPS;                    // Подождем
12 LCD_PUL_E;               // Дрыгнем стробом
13 NOPS;                    // Подождем
14
15 LCD_OFF_CS1;             // Выключим
16 LCD_OFF_CS2;             // выбор кристалла
17 }
18
19 // Пишем данные в контроллер
20 void LCD_WR_DATA(u08 cmd, u08 CSC)
21 {                        // Поднятие сигналов
22 LCD_SET_DAT;             // Данные
23 LCD_SET_W;               // Запись
24
25 ON_CS(CSC);              // Выбор чипа
26
27 LCD_DATA_INS(cmd);       // Данные на шину данных
28
29 NOPS;                    // Подождем
30 LCD_PUL_E;               // Дрыгнем стробом
31 NOPS;                    // Подождем
32
33 LCD_OFF_CS1;             // Выключим выбор
34 LCD_OFF_CS2;             // Чипа.
35 }
36
37 // Чтение данных из байта, адрес которого уже должен быть установлен. Надо только
38 выбрать
39 // контроллер.
40 u08 LCD_RD_DATA(u08 CSC)
41 {
42 u08 outv;
43                          // Выставляем линии управления
44 LCD_SET_DAT;             // Данные
45 LCD_SET_R;               // Чтение
46
47 ON_CS(CSC);              // Выбираем чип (только один!)
48
49 NOPS;                    // Ждем
50 LCD_PUL_E;               // Дрыг стробом - пустое чтение, активация защелки
51 NOPS;                    // Ждем
52
53 LCD_UP_E;               // Строб вверх
54 NOPS;                    // Ждем
55 outv = LCD_DATA_PIN;     // Контроллер выдал данные на шину. Читаем их
56 LCD_DN_E;               // Строб вниз
57
58 LCD_OFF_CS1;             // Все свободны!
59 LCD_OFF_CS2;
60
61 return outv;             // Возвращаем считанное
```

```
}
```

Чтение слова состояния делать не стал. Т.к. дисплей работает весьма шустро, что на круг ожидания можно не уходить, а просто подождать несколько тактов. Собственно этих трех функций уже достаточно для работы :) Остальное все свистоперделки и удобства.

Вроде заливки экрана:

```
// Заливка экрана
void LCD_FILL(u08 byte)
{
        for(u08 i=0; i<8; i++)                  //Перебирая страницы X от 0 до 7
        {
        LCD_WR_COM(0xB8+i,(1<<CS1|1<<CS2));    //Льем параллельно в оба чипа. 0xB8 задает тип операции (адрес страницы)

        for(u08 j=0; j<64; j++)                         // Задаем адрес Y от 0 до 64
                {
                LCD_WR_COM(0x40+j,(1<<CS1|1<<CS2));             // Записываем адрес Y в контроллер.
                LCD_WR_DATA(byte,(1<<CS1|1<<CS2));              // Записываем в байт с адресом XY  нужное значение
                }
        }
}
```

Или установки пикселя. Причем можно загонять его в трех режимах. Просто установить, стереть или инвертировать. Координаты задаются глобальные, а вычисление адреса идет в функции:

```
// Процедура установки пикселя. Т.к. пиксель часть байта, то надо сначала вычислить
// Контроллер, потом страницу и нужный байт. Считать этот байт. Изменить в нем только один,
// нужный, бит и вернуть его на место.
// На входе координаты и режим обработки пикселя (вкл, выкл, переключение)
void PIXEL(u08 x,u08 y,u08 mode)
{
u08 CSS, row, col, byte;
u08 res,read;

if(y>63)                        // Проверяем в каком контроллере искомый пиксель
        {
        CSS = 1<<CS2;           // Если У больше 63 значит во втором. Выставляем 2й
        col = y-64;             // И отнимаем смещение, чтобы не мешалось.
        }
else
        {
        CSS = 1<<CS1;           // Иначе контроллер у нас первый.
        col = y;                        // А смещения нет.
        }

row = x>>3;                     // Делим X на 8, чтобы получить номер страницы.
byte = 1<<x%8;                  // А остаток от деления даст нам искомый бит, который мы задвигаем
                                // и получаем нужную нам битмаску для модификации байта.

SET_ADDR(row,col,CSS);          // выставляем адрес
read = LCD_RD_DATA(CSS);        // Читаем данные (адрес при этом ++ аппаратно).

switch(mode)                    // В зависимости от режима
        {
        case 0:                 // Clear
                        {
```

```
34                      res = read & ~byte;     // Накладываем сбрасывающую (NOT) маску
35                      break;
36                      }
37          case 1:     // Invert
38                      {
39                      res = read ^ byte;      // Накладываем инвертирующую (XOR) маску
40                      break;
41                      }
42          default:            // Set
43                      {
44                      res = read | byte;      // Накладываем устанавливающую (OR) маску
45                      break;
46                      }
47          }
48 SET_ADDR(row,col,CSS);         // Повторно выставляем адрес. Т.к. чтение его исказило.
   LCD_WR_DATA(res,CSS);         // Вгоняем туда результат нашей модификации.
   }
```

Но работа с пикселями медлення штука. Это же каждый надо считать, изменить, закинуть обратно. Куча мороки. На обновление всего экрана в попиксельном режиме уйдет прорва времени. Я подсчитывал, при полной загрузке проца только на это, на заливку попиксельно всего экрана уходит чуть ли не треть секунды.
Куда эффективней работать с дисплеем блоками. Т.е. берем и последовательно записываем сразу куски страниц, на глубину в несколько байт. Например, таким образом удобно рисовать текст. Правда при попиксельном выводе, когда текст попадает между страницами получается неудобно, приходится делать вдвое больше работы. Но все равно намного быстрей чем попиксельно. На порядок!

```
1  // Запись сразу блока. Удобно для вывода строк или картинок, постранично.
2  // На входе страница X и колонка Y. А также длина блока и адрес откуда брать данные.
3  u08 BLOCK(u08 x,u08 y, u08 len, u16 addr)
4  {
5  u08 CSS,i,col;
6
7  if(y>63)                    // Сначала вычисляем нужные нам сегмент (чип)
8          {
9          CSS = 1<<CS2;
10         col = y-64;
11         }
12 else
13         {
14         CSS = 1<<CS1;
15         col = y;
16         }
17
18 SET_ADDR(x,col,CSS);         // Ставим адрес
19
20 // А дальше в цикле гоним байты. Не забывая увеличивать адрес, чтобы была выборка из
21 // памяти. Счетчик, чтобы не сбиться со счета. И номер колонки, чтобы не вылезти за
22 границы
23 // И вообще понимать где мы находимся.
24 for(i=0;i!=len;i++,addr++,col++)
25         {
26         if(64==col)                 // Попутно проверяем за границы выхода из сегмента
27                 {
28                 if(CSS == (1<<CS2))     // Если случилось, и у нас второй сегмент, т.е.
29 конец
30                         {               // страницы (уже второй в этой строке)
31                         return 128;     // выходим с кодом ошибки (код больше разрешения
32 экрана)
33                         }
34
35                 col=0;                  // Иначе же обнуляем счетчик колонок. И
36 переключаем банку
37                 CSS = 1<<CS2;           // Выбрав второй сегмент экрана
```

```
38                SET_ADDR(x,col,CSS);    // И выставив новый текущий адрес
39                }
40
          LCD_WR_DATA(pgm_read_byte(addr),CSS); // Пишем туда данные прям из флеша (таблица
   символов).
          }
   return y+len;                          // Возвращаем координату увеличиную на размер
   блока.
   }
```

Блочный вывод легко превращается в текстовый вывод. Надо лишь совместить таблицу символов с печаталкой блоков. И гнать данные из флеша напрямую.

```
   // Процедура вывода строки. На входе строка, и координаты. X в страницах, а Y в точках.
1  void LCD_putc(u08 x,u08 y,u08 *string)
2  {
3  u08 sym;
4
5  while (*string!='\0')          // Пока первый байт строки не 0 (конец ASCIIZ строки)
6          {
7          if(127<y)              // Проверяем за границу выхода за экран. Если вылезаем
8                  {              // Вот тут, кстати, можно поиграть с числом, чтобы не
9  рубило последний символ.
10                 y=0;           // То обнуляем координату точки.
11                 x++;           // Выставляем следующую строку
12                 if(x>7) break; // Если экран и вниз кончился - выход.
13                 }
14
15         sym = *string-0x20;    // Вычисляем из ASCII кода смещение в таблице символов.
16                                // Для русского языка и цифр надо условия добавить. Т.к.
17                                // таблица там не полная, не 255 байт.
18
19         y = BLOCK(x,y+1,5,(u16)symboltable+sym*5);          // Закатываем этот блок.
20         string++;                                          // Не забывая увеличивать
21 указатель,
22         }
   }
```

Для еще большего ускорения можно немного оптимизировать функции ввода вывода. Убрав из хвостов возврат сигналов в первоначальное положение. У меня они больше для красоты. Да чтобы на экране анализатора было лучше видно :)

Ну и чтение-модификацию-запись делать тоже большими блоками. Загоняя, скажем, страницу в буфер, правя его там и быстро выгружая обратно.

Ну и, традиционно, архив с примером. Сделано все на ATmega16 под WinAVR GCC на Pinboard [1]

Скачать пример кода [4]

AVR Studio 4.19 и AVR Toolchain

Обновился я тут недавно до AVR Studio 4.19 и спустя некоторое время обнаружил, что все проекты, что были на Си, отказываются компилироваться. Либо компилятся, но не отлаживаются. После недолгого выяснения и теребления народа из сообщества [1]выяснилось, что WinAVR как самостоятельный проект ныне не существует, а полностью перешел под крыло Atmel и ныне зовется **AVR Toolchain**. Ставить его нужно отдельно с сайта Atmel [2], предварительно снеся старый WinAVR, поверх студии 4.19. Ставится он теперь в дебри папки студии. В целом ничего не изменилось, по крайней мере все старые проекты скомпилились без проблем.

Чтобы вам не париться с регистрацией на сайте Atmel я бросил пару файликов:

- **AVRStudio 4.19** [3]
- **AVR Toolchain 3.3.0** [4]

FT2232D и AVR. Прошивка и отладка по JTAG

AVRdude Bitbang
Поскольку я привык к связке avrdude+sinaprog [1] то менять ее на что либо совершенно не хотелось. Поэтому в первую очередь я решил проверить, а можно ли юзать FT2232 в том же режиме, что и FT232RL [2]. И пнул дудку в адрес включенной FTDI. Она мне отозвалась, что мол устройство ft0 найдено и работает, но вот контроллер не подключен. Ожидаемо. Осталось только выяснить соответствие выводов у FT2232 и написать конфиг для avrdude.

Все оказалось проще чем я думал. С точки зрения avrdude микруха FT2232 представляет собой всего лишь две FT232R которые она видит как ft0 и ft1 (ну либо другие ft в зависимости от числа FTDI микросхем воткнутых в систему). А дальше все оказалось совсем элементарно — выводы шины ADBUS принадлежали интерфейсу ft0, а BDBUS устройству ft1. Логично, чо.

Теперь берем конфиг для avrdude и руководствуясь теми же соображениями, что и для FT232RL [2] присваиваем выводы. Прям по порядку, как нам удобней, предварительно создав еще одну секцию и обозвав наш программатор 2ftbb:

```
1  #FTDI_Bitbang
2  programmer
3    id    = "2ftbb";
4    desc  = "FT232R Synchronous BitBang";
5    type  = ft245r;
6    miso  = 5;  # ADBUS 5
7    sck   = 6;  # ADBUS 6
8    mosi  = 4;  # ADBUS 4
9    reset = 7;  # ADBUS 7
10 ;
```

Почему в таком порядке? Да просто удобно — у ATMega16 выводы программирования идут по корпус подряд RST, SCK, MISO, MOSI и сделав их в том же порядке на выводах FTDI мы можем простым плоским шлейфом соединить один к одному и прошить микросхему (подав естественно питание и землю)

Кстати, о питании. Я в изначальной разводке платы [3] забыл вывести на колодку питание с USB. Т.е. программатор то запитан, а вот таргет схема должна иметь свой источник питания. Впрочем, это легко обходится. Достаточно сделать хитрый тройной джампер, соединяющий вместе все три пина переключателя VCCIO_PWR_SEL в этом случае на пине CPU_PWR появляется напряжение с USB. Я этот джампер сделал из огрызка PBS линейки, закоротив у него три вывода спайкой. Если будете повторять схему адаптера, то подумайте над этим моментом, может джампер вынесете какой.

Набрасываем лапшу на контроллер и шьем. Шьется на ура. Собственно так и задумывалось :)

Также существует масса альтернатив программаторов AVR на базе FT2232. Например, широко известный **AVReal** работает с точно такой же схемой [4]. Разве что распиновка немного иная, но это не большая проблема. Кроссплатформенный, очень быстрый и простой. Консольный, но есть оболочка от сторонних авторов.

AVR Happy Jtag2 — Порция щастья!
Ковырясь по инету, чтобы найти что бы еще повесить на FT2232 я нашел и немного счастья в лице интересного проекта Happy Jtag от Словенских разработчиков [5].

Суть простая — FTDI обеспечивает связь между отлаживаемым контроллером и программой, а программа косит под JTAG ICE II и подбрасывает студии виртуальный порт, который та наивно принимает за JTAG ICE II. Опаньки!

Также сим девайсом можно шить ряд популярныйх контроллеров, как простым программатором.

Схема включения простейшая:

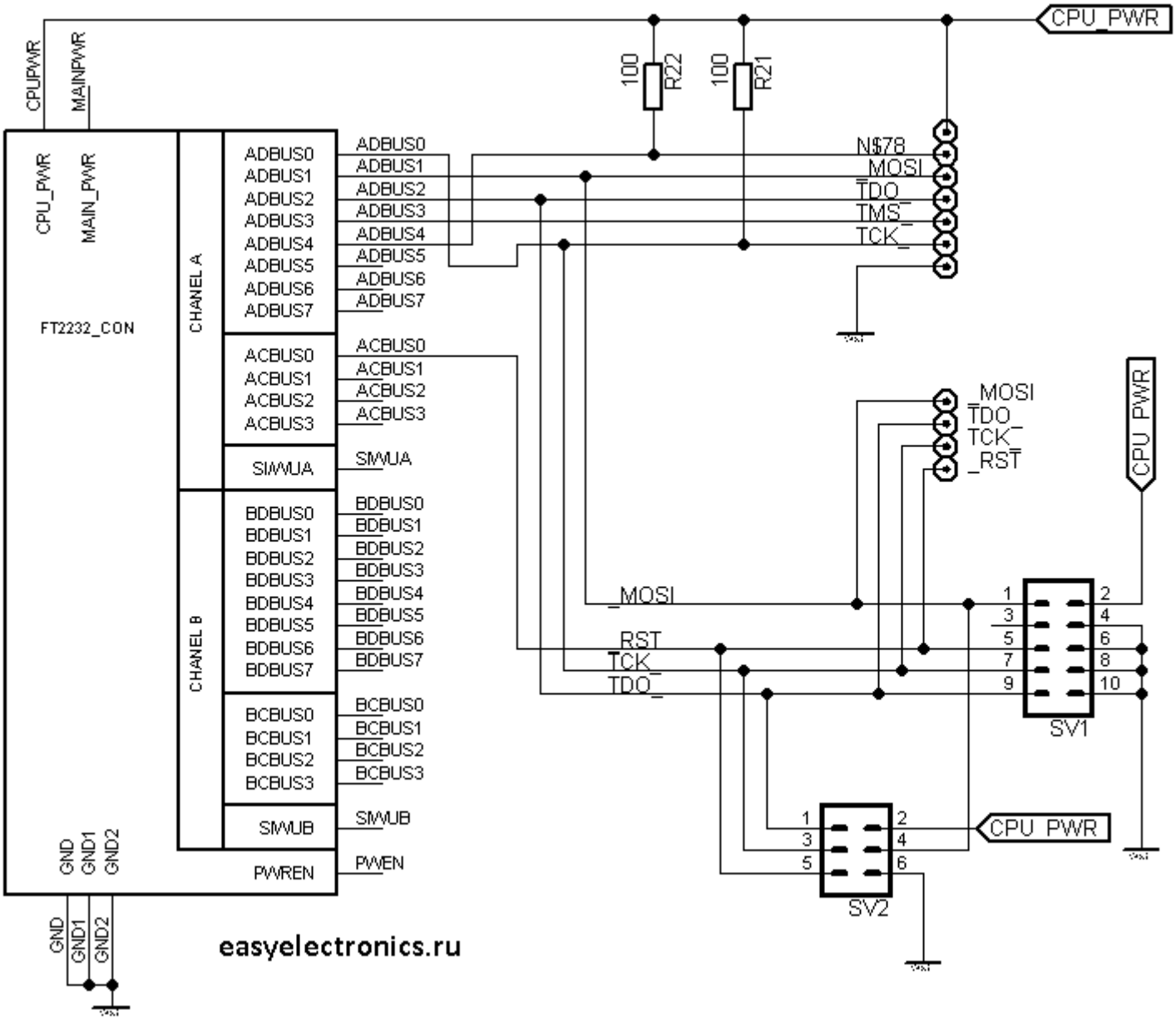
- ADBUS0 = TCK (надо еще подтянуть его через 1.5к к питанию)

- ADBUS1 = TDI
- ADBUS2 = TDO
- ADBUS3 = TMS
- ADBUS4 = RESET (надо еще подтянуть его через 1.5к к питанию)

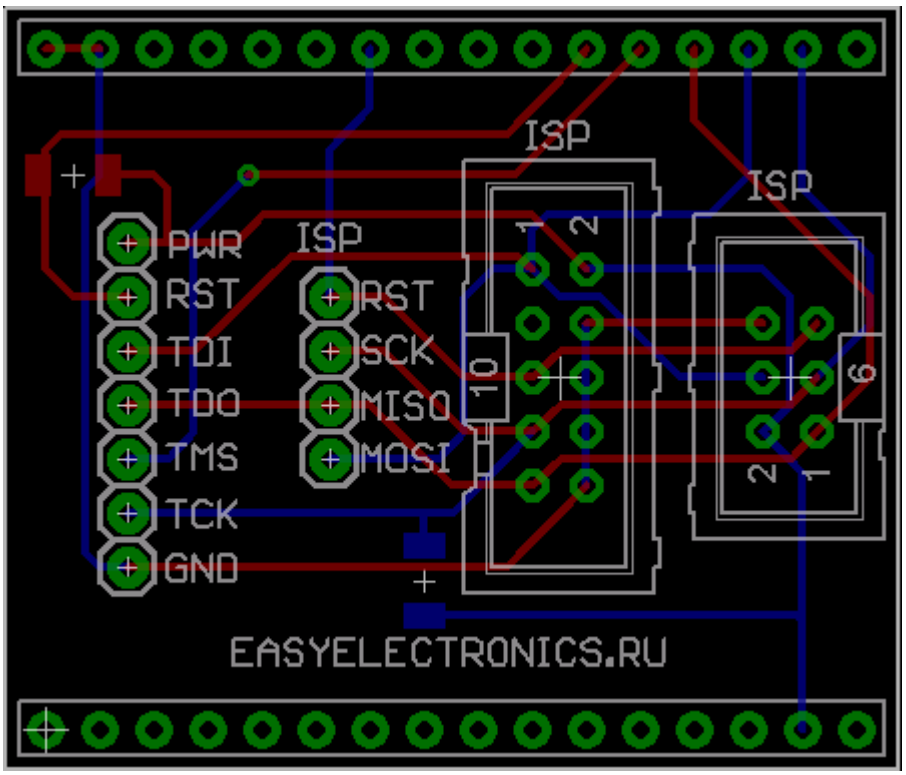
Для прошивки по SPI включение следующее:

- ADBUS0 = SCK (надо еще подтянуть его через 1.5к к питанию)
- ADBUS1 = MOSI
- ADBUS2 = MISO
- ACBUS0 = RST

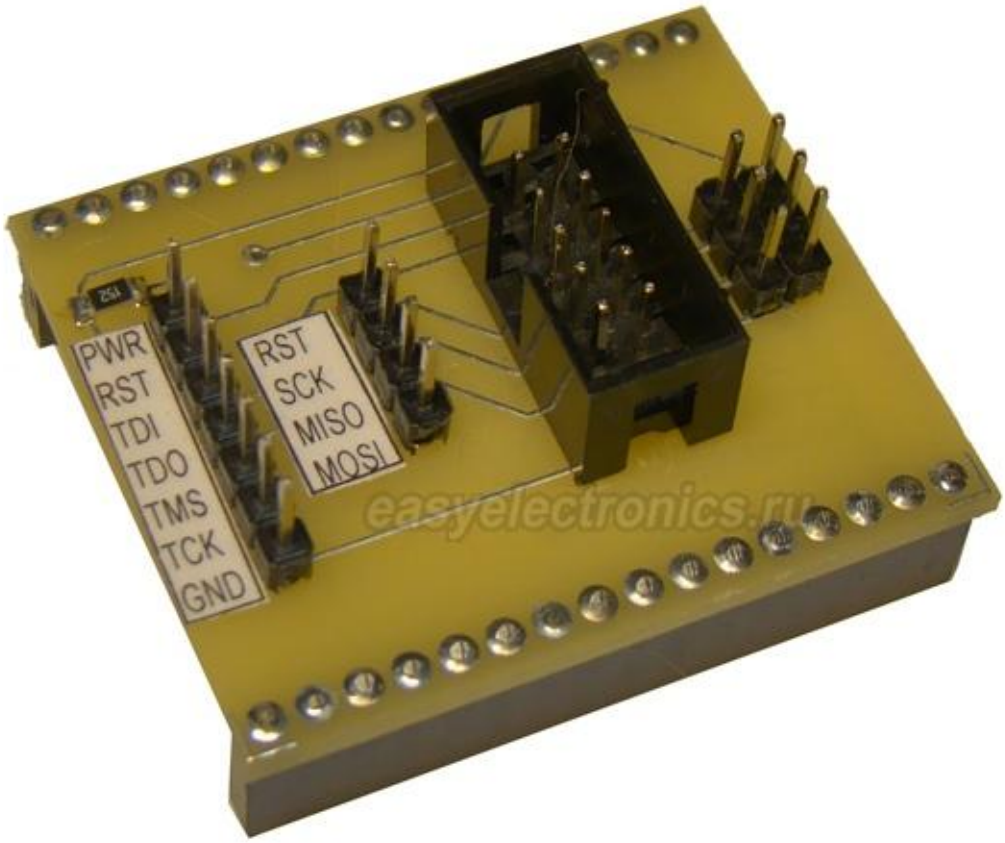
Схемка как и описано. Слева это не FT2232, а сразу весь модуль [3]. Чуть попозжа я его выложу его Eagle библиотеку, как заготовку для удобного изготовления переходников.



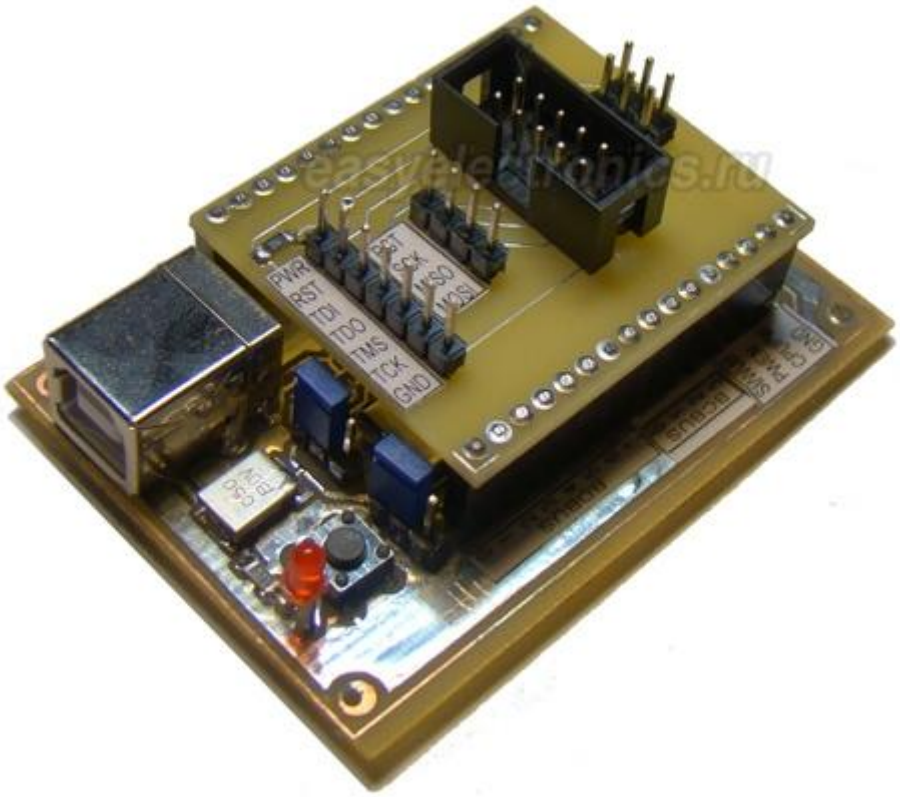
Я под это дело сваял небольшую платку. Запихав в нее разьемы всех используемых мною видов. Плата двусторонка, т.к. пихалась в общий проект который делали на заводе, поэтому я не заморачивался. Даже переходная дырка есть :)



И вот такая платка:



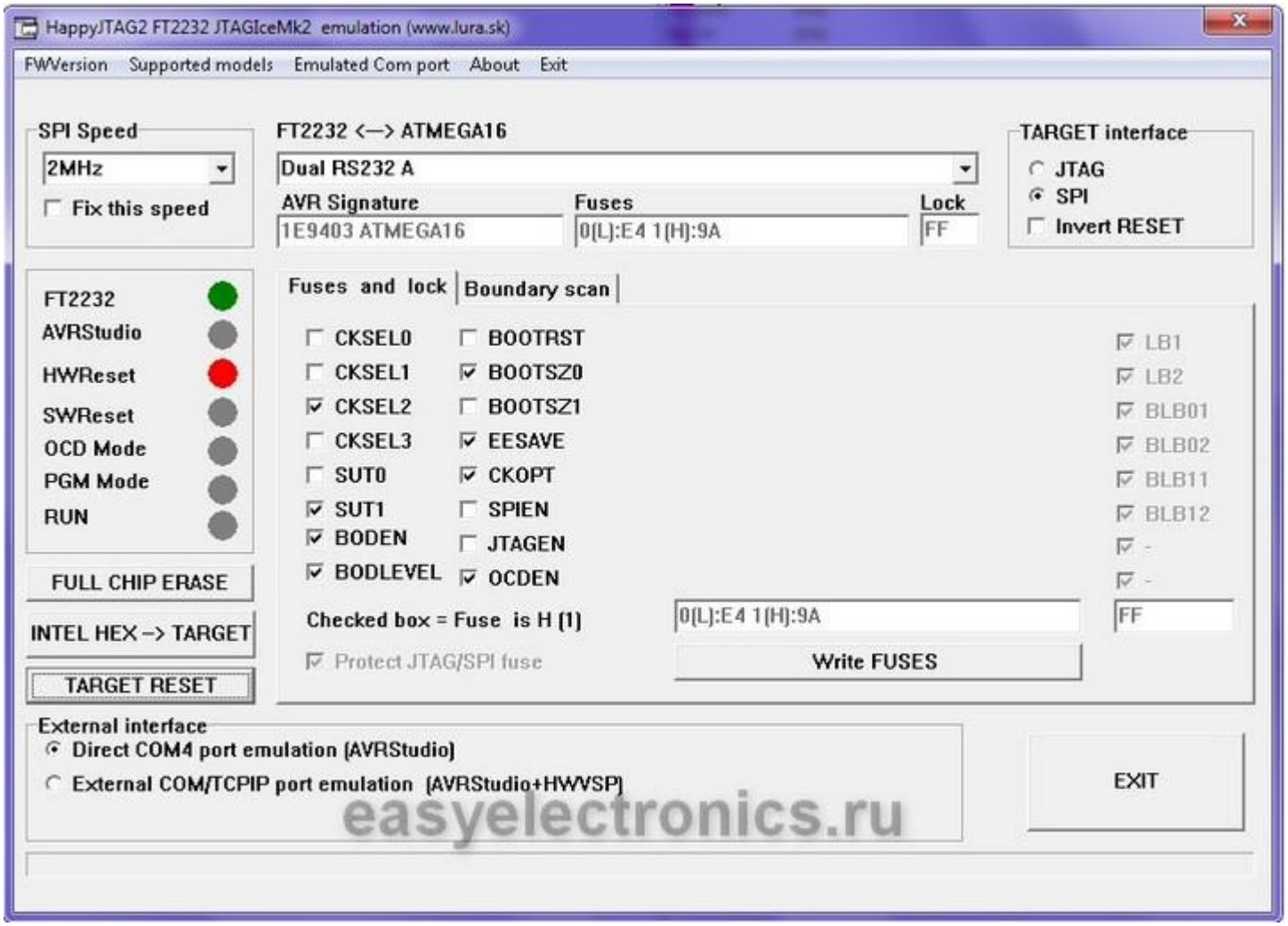
Сажается это все сверху на нашу конструкцию:



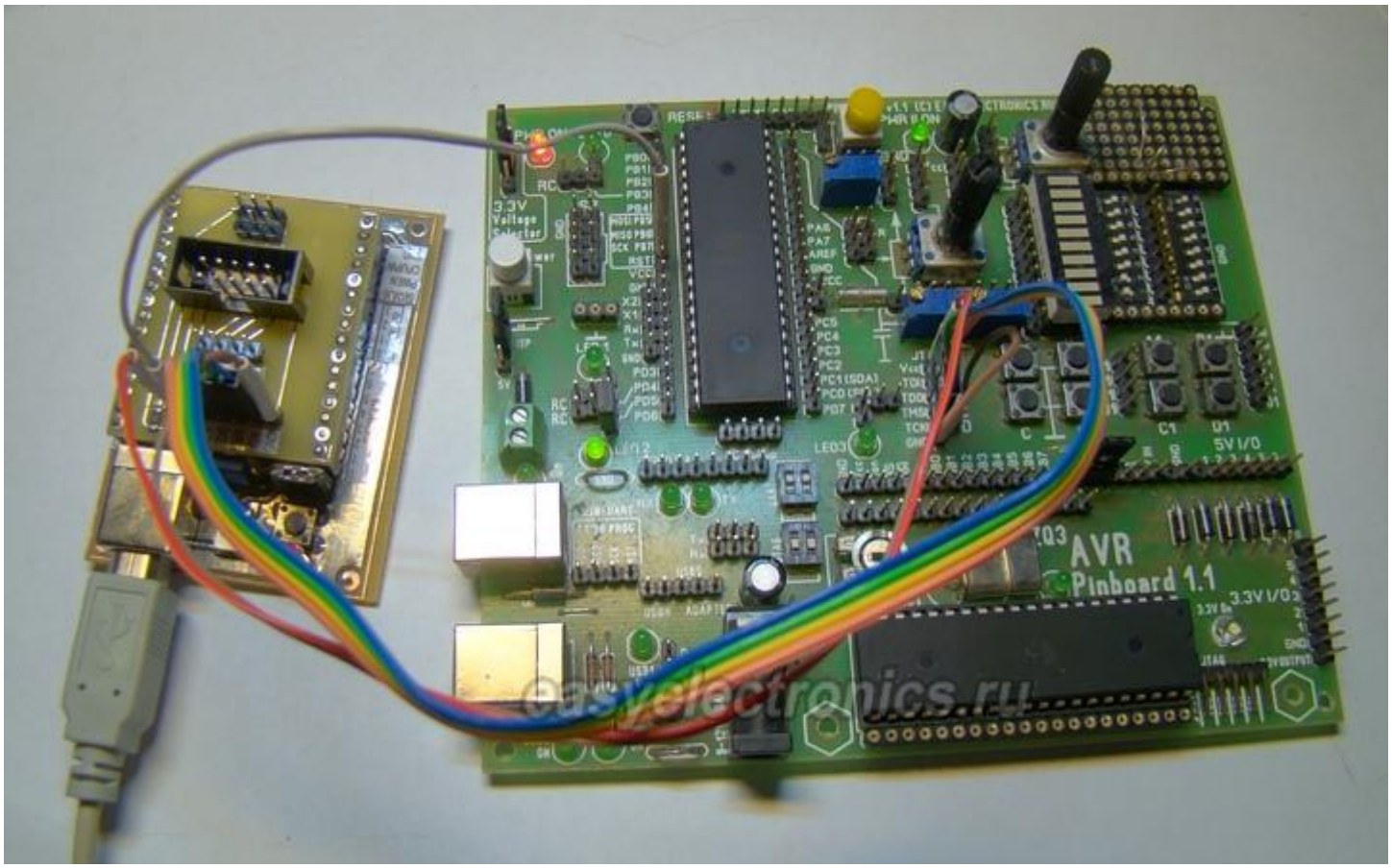
Дальше, если нам надо прошиться, то цепляем обычный шлейфик с IDC10 разьемом и соединяем таргет с нашим адаптером.

И запускаем программку, запускать надо от имени администратора (если в Win7 или Vista) иначе ничего нормально работать не будет.

В качестве интерфейса выбираем SPI и можно шить фузы (они, кстати, не инверсные. Как по даташиту. Внимательней!) и заливать HEXы в память.



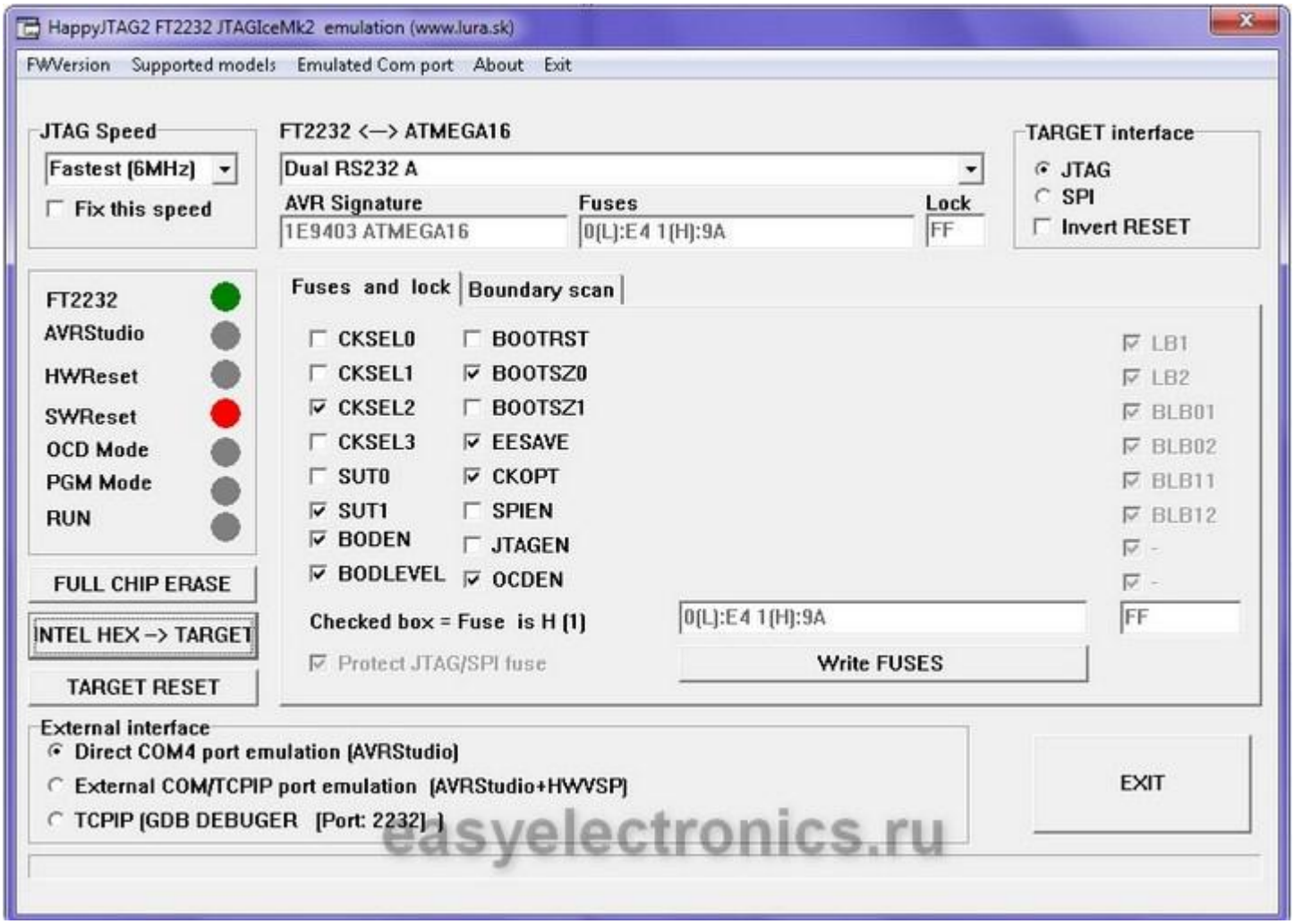
Но прошивка это не интересно. Отладка куда веселей. Так что соединяем JTAG лапшу в виде пяти проводков (TCK, TDI, TMS, TDO, RST) и питания, если нужно:



Стандартный JTAG разьем я делать не стал. Он слишком громоздкий и в любительских поделках его ставить обычно жалко места.

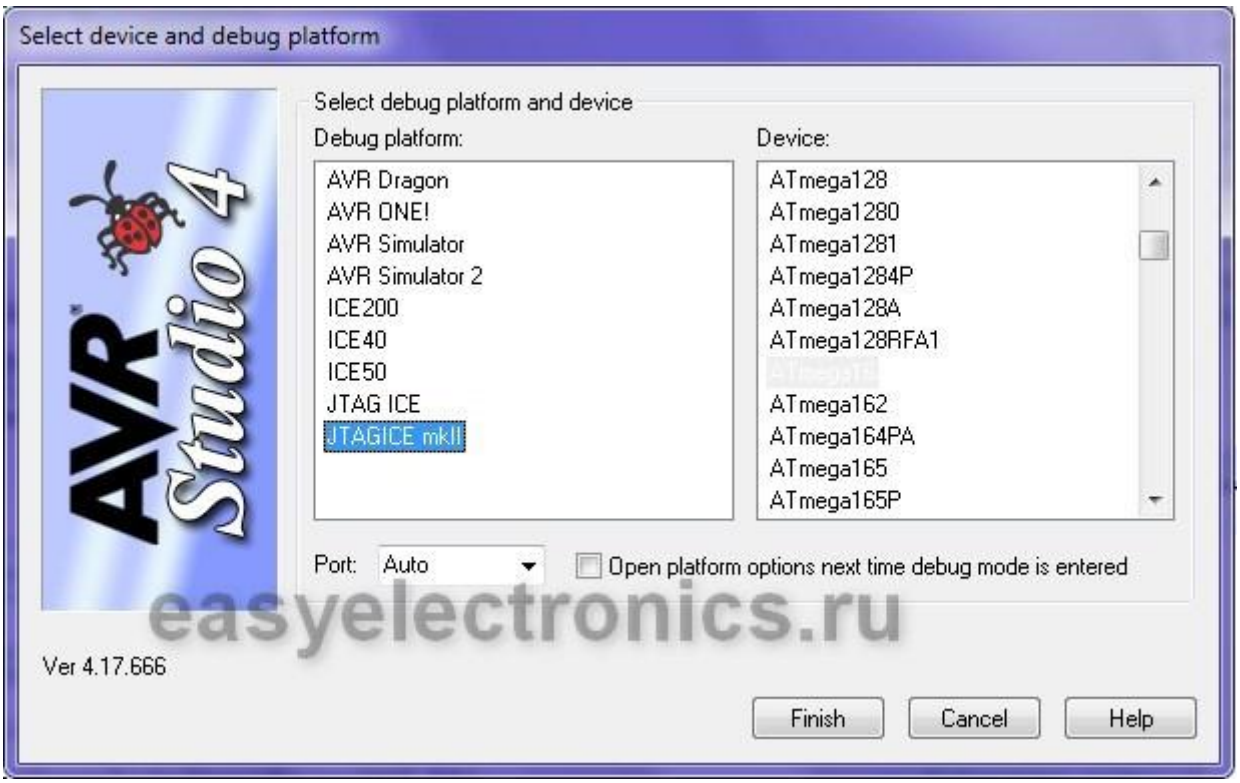
Запускаем Happy JTAG2 с правами администратора, выбираем в качестве интерфейса JTAG, а в качестве порта COM4 (без запуска под админом он не сможет создать виртуальный COM4 порт). Естественно надо предварительно обеспечить, чтобы в системе не было COM4 и тем более он не был занят :)))) Думаю это понятно. Иначе будет ошибка и нифига не заработает.

У меня на COM1 железный порт мамы, COM2 и 3 создала FT2232, а 4й сделал Happy JTAG.

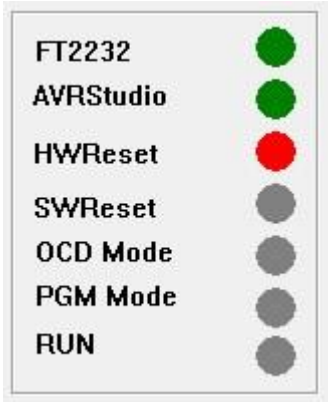


На панели лампочек должна загорется зеленая лампочка напротив FT2232 т.е. у нас определилась микруха, а в поле AVR Signature должен определиться контроллер. Появятся фуз биты и возможность прошить контроллер через JTAG. Естественно интерфейс JTAG должен быть включен фуз битом JTAGEN.

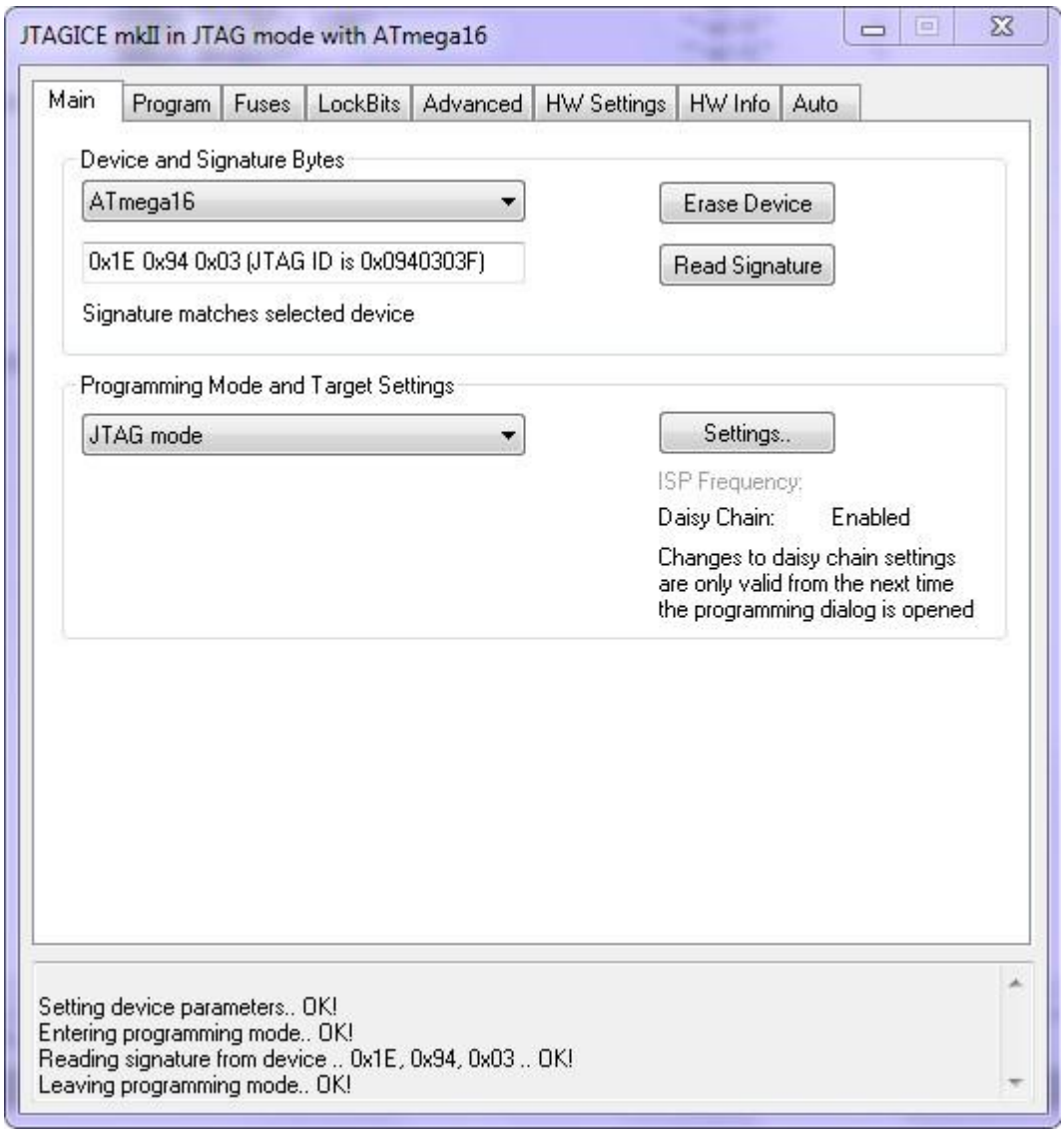
Теперь запускаем студию (рекомендую 4.19 или около того [6]) и выбираем там в свойствах нашего проекта контроллер и в качестве адаптера JTAG ICE II



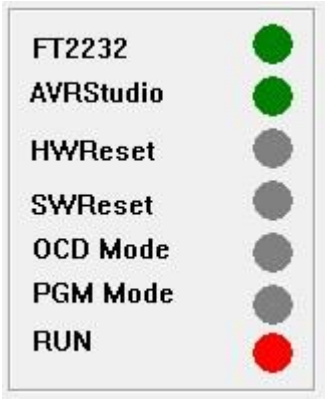
На панели лампочек зажгется индикатор подцепленной студии:



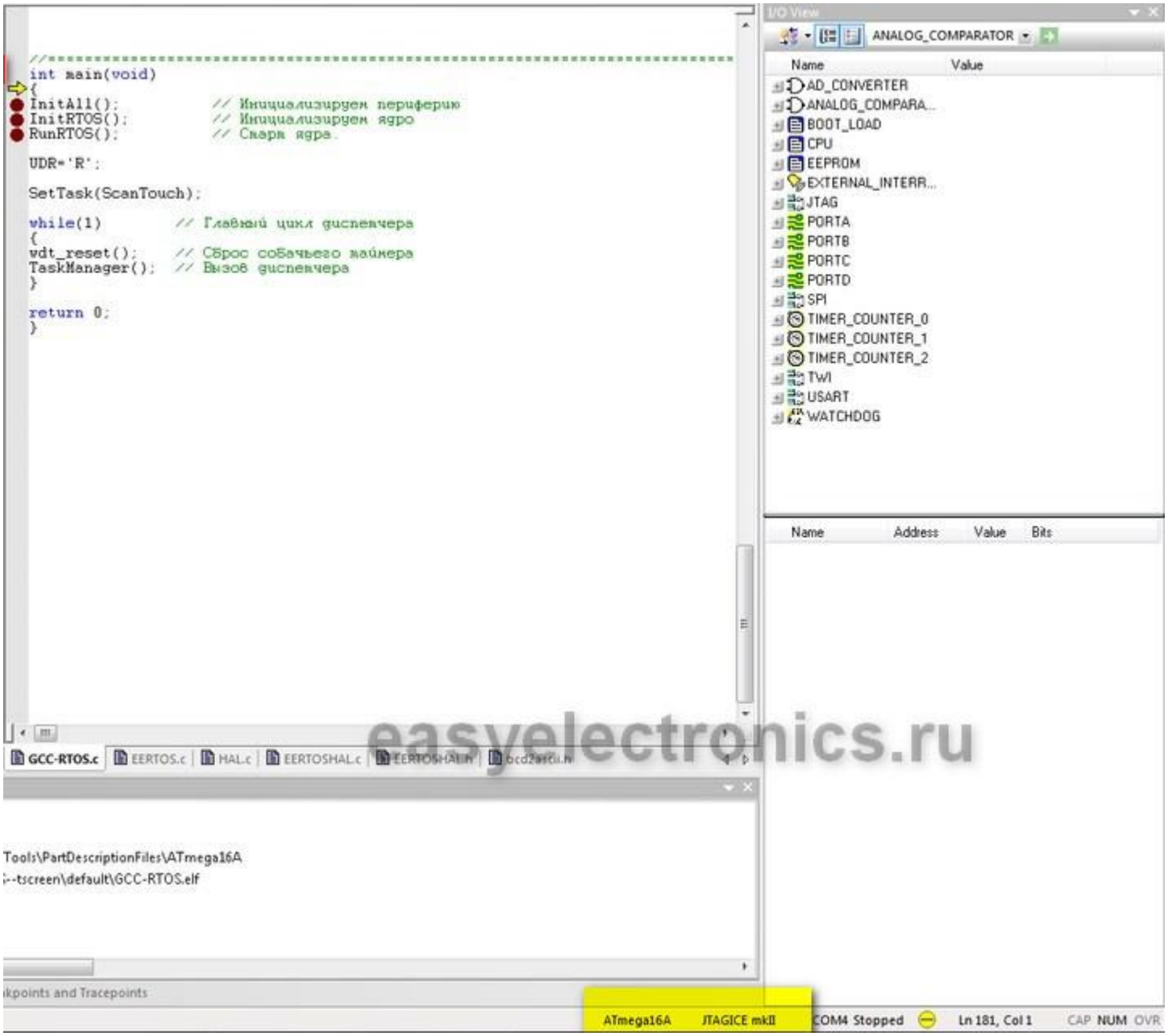
А в самой студии станет доступен в качестве адаптера JTAG ICE II



Теперь можно запускать процесс отладки и шагать по программе. Работает вполне стабильно. У меня пока серьезных зависаний не наблюдалось.



При этом на панели горит лампочка, сигнализирующая о том, что процесс идет:



Работает не быстро. Не сказать, что сильно быстрей чем первый ICE, есть некоторые проблемы с ассемблерными прогами (!) — на них часто выводит FFFF вместо команд. С Сишными работает сносно. Возможно это из-за особенностей линковки. Зато можно ставить большее количество брейкпоинтов (я ставил около десятка, работает). Еще бывает теряет связь и отваливается, впрочем, первый ICE тоже этим грешил постоянно. Но как халявный вариант — почему бы и нет?

Да, любители AVR Studio 5 обламываются. Т.к. в этой дивной программе атмеловцы оторвали нахрен все старые девайсы, прекратив поддержку ICE I и всех COM программаторов. Покупайте JTAG ICE III :)
Что в свете агрессивного наступления STM8 и STM32 с копеечными (а часто и вовсе бесплатными) средствами полноценной отладки выглядит как маразм граничащий с самоубийством.

Ну и напоследок, как обычно файлики

- **Happy JTAG2 (но лучше заглянуть на сайт авторов и взять там, может есть более свежая версия)** [7]
- **Печатки платки адаптера (пока в PDF)** [8]
- **Собственно сам главный модуль, на который все это одевается** [3]