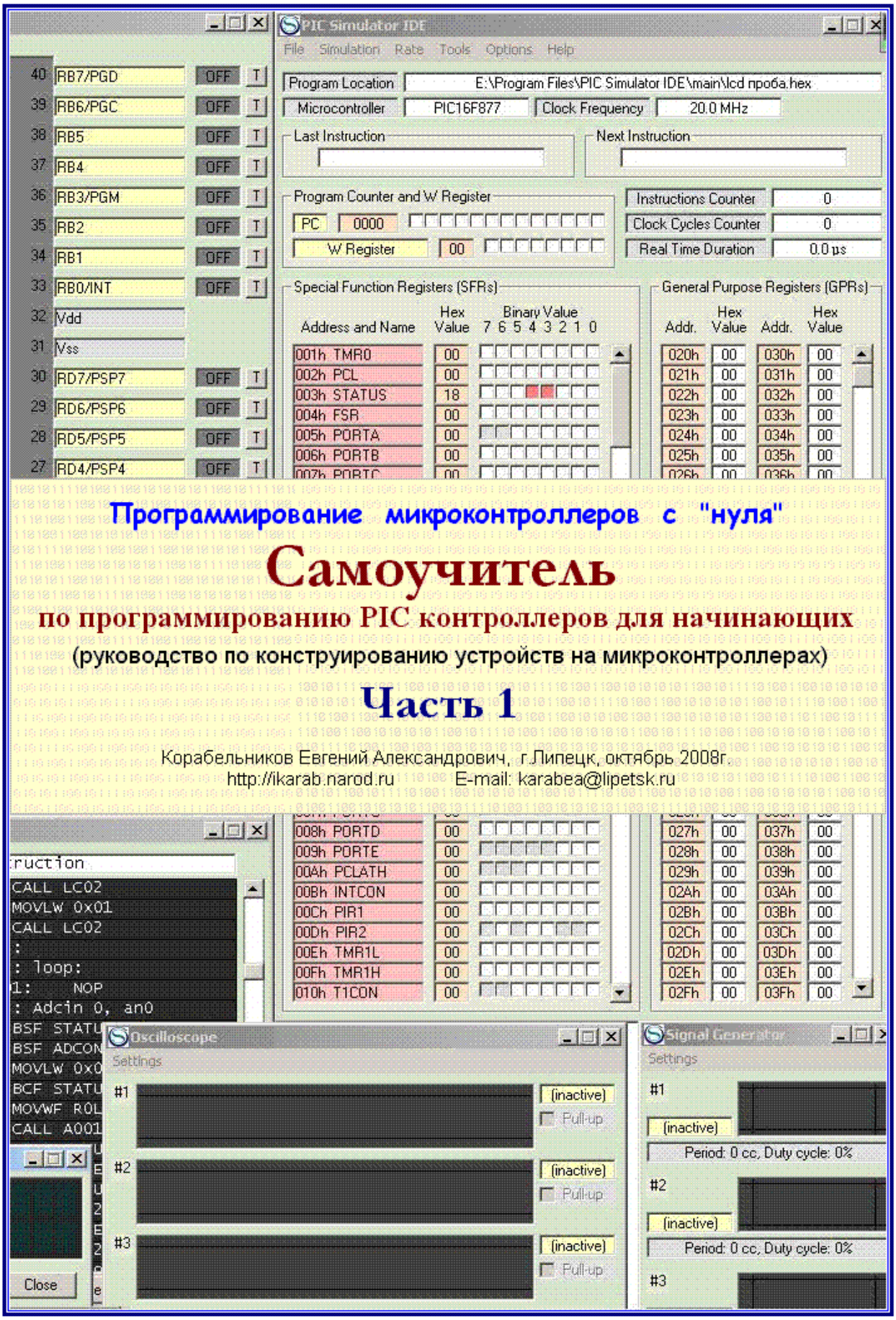


Программирование микроконтроллеров с "нуля"

Самоучитель

по программированию PIC контроллеров для начинающих

(руководство по конструированию устройств на микроконтроллерах)

Часть 1

Корабельников Евгений Александрович, г.Липецк, октябрь 2008г.
http://ikarab.narod.ru E-mail: karabea@lipetsk.ru

Часть 1
Оглавление

Введение 3

1. Готовим инструменты 5

2. Что такое микроконтроллер и как он работает 10

3. Система команд PIC16F84A 26

4. Что такое программа и правила ее составления. Пример создания программы автоколебательного мультивибратора. Директивы. 36

5. Интегрированная среда проектирования MPLAB IDE и работа в ней 45

6. Что дальше? 70

7. Пример создания программы (начало) 72

8. Пример создания программы (продолжение) 96

9. Работа в симуляторе. Отладка программы 107

10. Как отследить выполнение программы 121

11. Прерывания. Стек. Пример разработки программы с уходом в прерывания 134

12. Организация вычисляемого перехода. Работа с EEPROM памятью данных 160

13. Флаги. Работа с флагами. Как работает цифровой компаратор. Перенос и заем 182

14. Пример задействования флага C в трехбайтном суммирующем устройстве. Циклический сдвиг. Операция умножения 200

15. Введение в принцип построения подпрограммы динамической индикации. Косвенная адресация 222

16. Преобразование двоичных чисел в двоично-десятичные. Окончательное формирование текста подпрограммы динамической индикации 243

17. Принцип счета. Работа с таймером TMR0. Принцип установки групп команд счета в текст программы 262

.....

.....

Заключение 277

Дополнительная информация 279

Количество страниц: 287.

Введение

Эпиграф: программист, работающий в ассемблере, должен быть "властелином колец".
И еще: банальность это уставшая истина.
(поймете позднее)

В микропроцессорную технику люди приходят по-разному.
Лично я, до поры, до времени, не ощущал особой потребности в необходимости заниматься этим, пока, в один прекрасный момент, не понял, что начинаю не соответствовать времени.
То, что я наработал "до того", оказалось безнадежно устаревшим, а также "смешно смотрящимся" и на "фоне" современной элементной базы, и на "фоне" тех знаний, которые нужно иметь для того, чтобы работать с ней.
Кроме того, лично для меня, как-то не красиво и не достойно было "питаться объедками с царского стола", если есть возможность "за него сесть на правах полноценного участника трапезы".
Нужно было выбрать: либо "сложить лапки" и перейти в категорию постепенно "вымирающих" (дисквалифицирующихся), либо заняться этими "страшными и ужасными" микроконтроллерами, которые все более напоминали "в каждой бочке затычку".
"Вымирать" совсем не хотелось, так что выбор был однозначным.
И тут началось нечто, что напоминало "передвижение по джунглям".
Информационный "бардак" в этом "секторе" оказался настолько впечатляющим, что "волосы встали дыбом".
А куда деваться? "Отступать-то некуда, позади Москва".
Кстати, точно в таком же положении находятся сейчас многие люди (знаю по письмам), для которых "въезд" в микропроцессорную технику стал не то что какой-то "блажью", а самой натуральной жизненной необходимостью, что вполне понятно, ведь м/контроллеры входят в состав практически любой более или менее современной, малогабаритной (и не только) аппаратуры (а "чем дальше в лес, тем больше дров"...).
Чего я натерпелся, знает только один Господь Бог: помощи никакой и пришлось рассчитывать только на свои силы.
После всех этих "мытарств", возник закономерный вопрос: "Это что же такое получается? Неужели каждый, кто вознамерится "посягнуть" на эти "железяки", должен обязательно "разбивать свой нос в кровь", водя им по "батарее"?
Неужели нельзя без этого обойтись или, по крайней мере, сделать этот процесс не столь болезненным?
Прикинул...
А ведь, ей Богу, можно!
Правда, придется "вспахать поле не паханное", но по совокупности причин, смысл в этом есть.
То, что Вы прочитаете в "Самоучителе...", есть итог указанного выше болезненного процесса, преподнесенный "на блюдечке с голубой каемочкой".
Принцип преподнесения информации - максимальная степень "разжеванности", так как "Самоучитель..." предназначен именно для начинающих.
Одна из главных бед начинающих программистов - отсутствие системности в восприятии информации и ее "передозировка", связанная с чрезмерным желанием побыстрее достигнуть желанной цели, без учета объективных факторов.
Такого рода желание, конечно же, похвально, но при отсутствии плановости, четко выраженных приоритетов и способности, на первых порах, сознательно ограничивать объем воспринимаемой информации только самой действительно необходимой, оно играет с человеком злую шутку.
В результате - "бардак" в голове, дезориентация в потоках информации и в худшем случае, сожаление о потраченном времени, хотя, по большому счету, все не так уж и суперсложно, как может показаться на первый взгляд.
Я вовсе не говорю, что это просто. Поработать придется, но и пугаться совсем не стоит, так

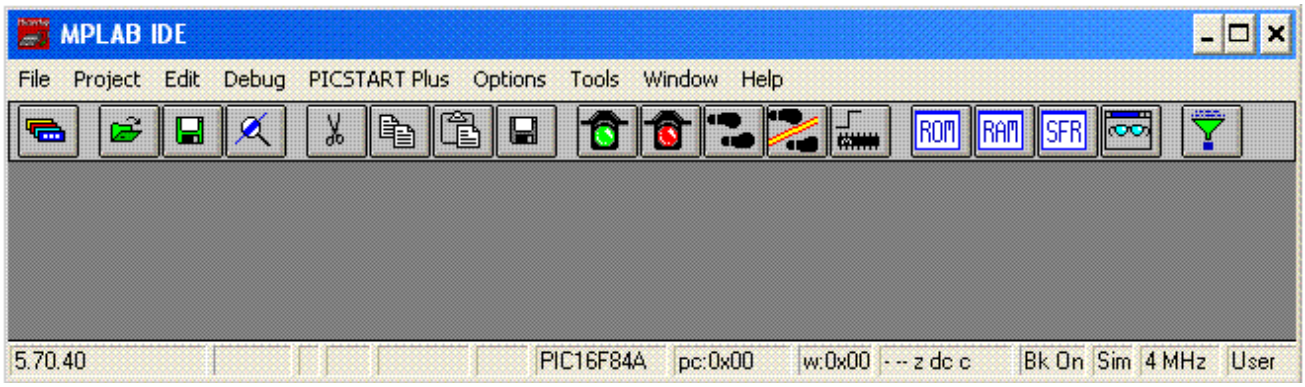
как "не так страшен черт, как его малюют".
Еще одна беда - недооценка огромного значения знания и умения применения на практике стратегии и тактики "мозгового штурма".
Хотя и любой "мозговой штурм" полезен, но "мозговой штурм" программиста, имеющего хотя бы элементарное представление о его стратегии и тактике, гораздо эффективнее и действеннее, чем "судорожные действия" программиста, который этих представлений не имеет. А ведь работа программиста это "сплошной мозговой штурм"!!!
Мозги есть у всех, а вот со стратегией и тактикой этого "штурма" имеются большущие проблемы. Можно ведь, с дуру, и "пулю схлопотать" (по сценарию типа "геройская смерть программиста").
В своей работе я исхожу из того, что мозги являются не только логической "машиной", но и "вместилищем личности".
Последнее либо явно недооценивается, либо вообще не берется в рассчет авторами подобных моему "творений", что есть огромнейший их просчет, сводящий на нет большую часть усилий.
Такого рода "однобокость", носящая массовый характер, в большинстве случаев, приводит к тому, что информация воспринимается обучаемым как логически изощренное, интенсивное (без чувства меры) и "беспросветное изнасилование автором его (обучаемого) мозгов", с целью "глумления" над низким уровнем его подготовки и прямого или косвенного понижения "микроконтроллерной" самооценки.
Конечно же, во многом, это не соответствует действительности, но что поделаешь, такова естественная, подсознательная, защитная реакция психики нормального человека на большой массив информации, к эффективной работе с которым она не готова.
Для того чтобы понять огромный вред такого подхода к обучению, вспомните про Афганистан или Чечню и про участь тех необстрелянных и психологически неподготовленных ребят, которых "бросили в эту мясорубку".
Я не желаю Вам такой участи, и по этой причине, в "Самоучителе...", предпринята своеобразная попытка постепенного "встраивания" нулей и единиц в личность (их "одухотворения") и формирования некой "идеологии офицера программных воск" ("боевого духа", "стержня"), без которой любая "война" (программирование есть чисто мужское и "хулиганское" занятие с названием "война со своей бестолковостью") проигрывается даже не начавшись и которая является главной основой любой эффективной "школы" обучения.
Сравнить мне не с чем, и поэтому я работаю на свой страх и риск.
Не судите меня строго, так как я работаю "с нуля" и "психологическим спецом" не являюсь.
Надеюсь на то, что другие авторы продолжат эту исключительно важную и "преступно" игнорируемую "психологическую тему". Хочется верить, что при чтении "Самоучителя...", Вы почувствуете, что такое доброжелательное и уважительное отношение к Вашему совсем не легкому труду (по себе знаю), а Ваше подсознание не будет выдавать сигналов SOS об "изуверском изнасиловании мозгов".
Отдельно обращаюсь к "хулиганам", "драчунам" и "задирам" (в обывательском понимании этих слов), "мозговая деятельность" которых явно выражена.
Вам не нужно объяснять, что значит "держать удар", "уклоняться", "давать сдачи", и "фингалы" Вас не смущают. По этой причине, программирование это, в первую очередь, Ваша "вотчина", где Вы можете славно "поохотиться".
В программировании, агрессивность есть достоинство, а не недостаток.
Здесь можно, от души, интеллектуально "помахать кулаками" (ограничений нет), плюс, "посворачивать шеи" многим достойным уважения "врагам" (ограничений нет), от чего, кстати, Вы однозначно получите большое удовольствие.
Итак, информация будет предоставляться в определенной последовательности и по принципу "от простого к сложному".
Прошу придерживаться этой последовательности и не переходить к следующим разделам без уяснения предыдущих. Дело это неторопливое и не требующее суеты.
Все "валить в кучу" не буду, "перенапряг" также постараюсь не создавать.
"Самоучитель…" рассчитан на начинающих, но при этом предполагается, что они, как минимум, знают основы цифровой техники.

Выражаю искреннюю признательность тем людям, которые помогли в работе над этим учебником.

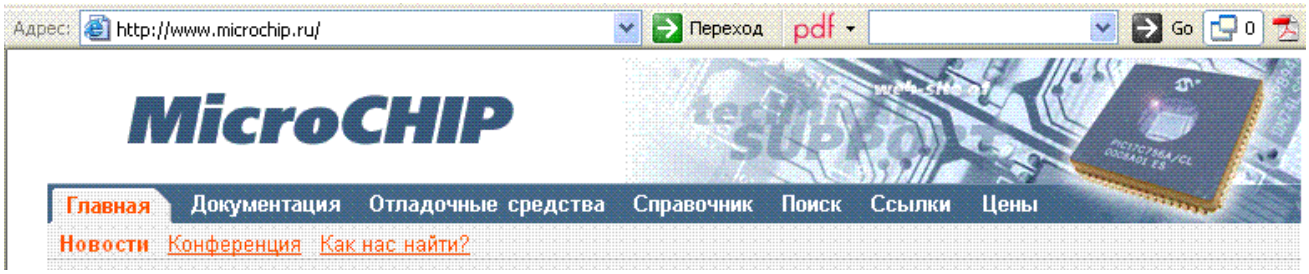
1. Готовим инструменты

Микроконтроллеры (и вообще все процессоры) изначально понимают только машинные коды, то есть некую совокупность нулей и единиц.
Те, кто представляет себе работу счетчиков, регистров, триггеров и т.д., сразу же поймет природу машинного кода.
Так как, среди электронщиков, таких людей большинство, то на мой взгляд, все они согласятся с такой аксиомой: машинные коды полезны в "малых дозах".
А вот когда начинаются "большие дозы" (сложные устройства с десятками корпусов м/схем), то "мозги начинают дымиться" даже у классных электронщиков, имеющих недюжинные способности.
В этом случае, самое неприятное заключается в том, что по мере роста схемотехнической сложности устройства, эффективность работы электронщика резко "падает".
И в самом деле, сил и средств вкладывается "море", а получается нечто не очень надежное, габаритное, сложное в изготовлении, энергоемкое и дорогое.
Чтобы "одним махом прихлопнуть" все эти проблемы, "яйцеголовые" и придумали сначала "большие" процессоры (то, что применяется в компьютерах), а затем и "маленькие", назвав их микроконтроллерами.
Внутри м/контроллера находится "набор" модулей, каждый из которых многофункционален.
Манипулируя весьма не слабыми возможностями этого "набора", можно реализовать миллионы разновидностей устройств.
Естественно, всем этим "хозяйством" нужно как-то "рулить".
Эта "рулежка" и есть то, что называется программированием.
Если речь идет о больших "массивах" машинных кодов, то программирования напрямую (в машинных кодах) и врагу не пожелаешь: удовольствия никакого, да, чего доброго, и в "психушку" попасть можно (есть исключения - люди с выдающимися способностями и гении).
Для того, чтобы обычные люди могли, без особого "напряга", заниматься составлением программ, придуманы различные языки программирования.
Смысл всех их заключается в замене машинных кодов словами, сокращениями слов, абревеатурами и т. д., то есть тем, что человеком легко и осмысленно воспринимается и чем он может комфортно оперировать при составлении текста программы.
Все эти "удобоваримые приятности", по окончании составления текста программы, переводятся в машинные коды одним "легким движением руки" (мозги программиста не задействуются).
Чтобы это "легкое движение руки" имело место быть, "яйцеголовые" придумали так называемую "интегрированную среду разработки".
Это есть набор программ, в котором программист работает с максимальной степенью комфорта, причем, по всему "массиву" решаемых им задач (включая и составление текста программы, и т.д. и т.п.).
Что, первым делом, нужно сделать, например, русскому, который попал в Англию и собирается там жить?
Выучить английский язык.
При "въезде" в программирование, нужно сделать то же самое (задача даже существенно проще).
"Проматерь" всех языков программирования - **ассемблер**.
Хотя он и считается самым простым, но слово "простой" относится прежде всего к набору его команд: количество их - минимально необходимое, и тем не менее, вполне достаточное для решения самых сложных задач, но не к комфортному восприятию их человеком.
Команды ассемблера являются либо сокращениями английских слов, либо набором первых букв английских словосочетаний, либо и тем, и другим.
Минимальный "джентльменский" набор ассемблера для ПИКов составляет 35 команд.
Реально же, наиболее часто, используются от 10 до 20 команд.
В дальнейшем, настройте себя просто на тупое заучивание (на первых порах) всей этой английской "абракадабры", типа зубрежки (я вообще не имею никакой склонности к иностранным языкам, но ничего, освоил), не такая уж это и сложная задача, заверяю Вас.
В дальнейшем, Ваше образное мышление и зрительная память Вам помогут.
А выучить ассемблер очень даже стоит по причине того, что он, может быть, и не очень "удобоварим", но именно на этом языке пишутся самые компактные по объему, быстрые и надежные программы, и по этой причине, серьезные программисты, работают

преимущественно в ассемблере.
Предупреждение: на этом этапе в ассемблер не лезть! Всему свое время. Пока достаточно общего представления (пусть "в мозгах уляжется").
Программы для ПИКов составляются преимущественно в ассемблере.
Даже если программа для них и составлена на языке более высокого уровня, то в конечном итоге, интегрированная среда разработки переведёт все в ассемблер.
Об интегрированной среде разработки (проектирования):
Она выполняет целый комплекс задач.
В ее специализированном текстовом редакторе, составляется текст программы.
Текст программы нельзя записывать в ПИК, так как он "понимает" только машинные коды.
Следовательно, нужно преобразовать текст программы, с языка ассемблер, в машинные коды.
То есть, необходимо так называемое ассемблирование (компилирование) исходного текста программы, которое производится все в той же интегрированной среде разработки.
Вот здесь-то начинающие обычно и путаются: словосочетание "ассемблирование исходного текста программы" означает не перевод исходного текста программы на язык ассемблер (текст программы уже написан на языке ассемблер), а наоборот, преобразование текста программы, написанной на языке ассемблер, в машинные коды, которые сначала соответствующим образом архивируются и помещаются внутрь специального файла с расширением (форматом) **.HEX** (для удобства хранения и транспортировки машинных кодов), а затем разархивируются из HEX-файла и принимают свой исходный вид в программе, обслуживающей программатор.
С помощью этой программы, машинные коды программы записываются в ПИК.
Приведенное выше словосочетание, используемое сплошь и рядом, безусловно, не является удачным.
Обратите на это внимание и всегда имейте ввиду, что оно не отражает смысла происходящего, хотя я и буду употреблять его далее, так как оно является стандартным.
Я рассказал только о двух основных функциях интегрированной среды разработки.
Ее возможности ими далеко не исчерпываются.
Интегрированная среда разработки для ПИКов называется **MPLAB**.



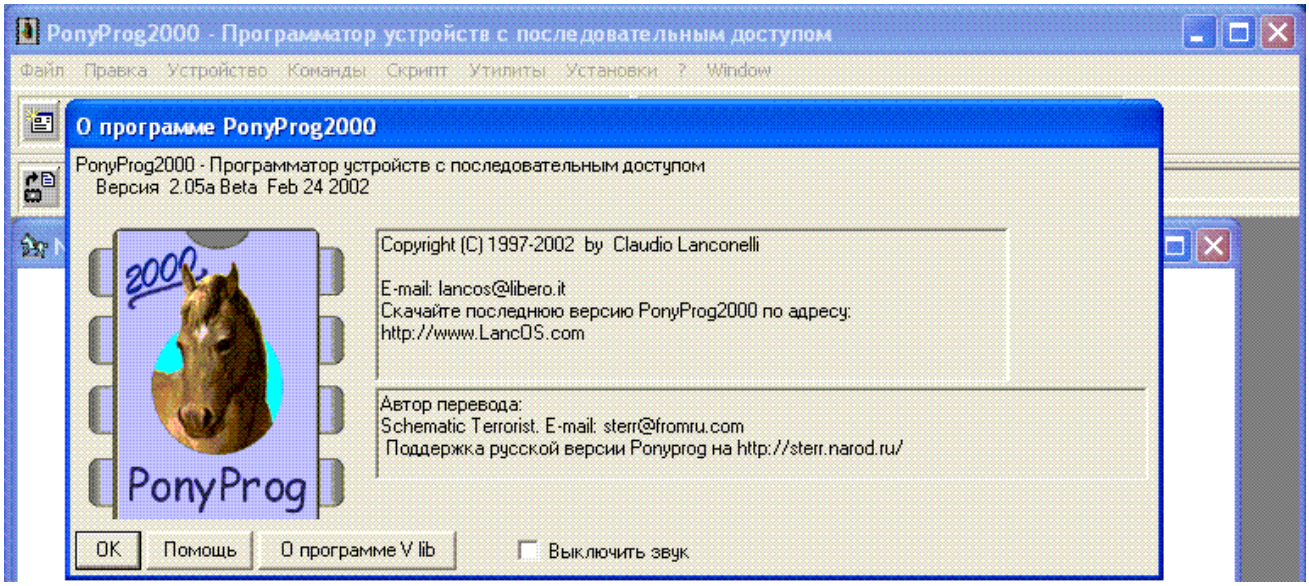
Эта программа (вернее набор программ) создана производителем ПИКов, то есть фирмой ***Microchip Technology Inc.***
В России, представителем этой фирмы является ООО "Микро-Чип", которое имеет в Интернете свой сайт технической поддержки на русском языке **http://www.microchip.ru** (кстати, на этом сайте, в разделе "Начинающим", есть ссылка на мой сайт).



Лично я пользуюсь версией **MPLAB 5.70.40**, что и Вам советую.
Это "старый, добрый конь, который борозды не испортит" и возможностей у него "выше крыши".
Главный недостаток этой версии - медленно работает (считает), но для начинающих, "реактивной" скорости и не нужно.
Главное ее преимущество - надежность работы.
В более поздних версиях, в той или иной мере, осуществлен обмен скорости на надежность, что иногда не есть хорошо.
В дальнейшем, я буду ориентироваться на версию 5.70.40.

Примечание: дистрибутив **MPLAB версии 5.70.40** (и еще 2 версии) имеется на компакт-диске. Закачивать дистрибутив **MPLAB** нужно в папку с английским названием (папка ***мои документы*** или ***рабочий стол*** не подойдут), иначе будете иметь проблемы.
Лучше всего организовать ее в папке ***Program Files*** диска ***C***.

Программа **MPLAB** является интегрированной средой разработки для ПИКов и содержит все необходимое как для написания и редактирования программы, так и для создания HEX-файлов, а также и для отладки программы.
Таким образом, необходимость в наличии отдельного текстового редактора для написания программы, отдельной программы - ассемблера для создания HEX-файла и отдельного отладчика программы (симулятора) отпадает, так как в **MPLAB** все это есть (и даже более того).
Установите **MPLAB** на свой компьютер, убедитесь, что она встала не "криво" и на время про нее забудьте, так как для того чтобы с ней работать, необходимо основательно подготовиться, чем в дальнейшем мы и будем заниматься.
Следующий шаг - **сборка программатора**, так как HEX-файл программы, созданный в **MPLAB**, необходимо "превратить" в машинные коды, которые и будут записываться в ПИК (так называемая "прошивка").
Ничего проще и надежнее чем программатор ***PonyProg***, я, на первом этапе, предложить не могу, хотя, безусловно, имеются и другие "достойные" программаторы.

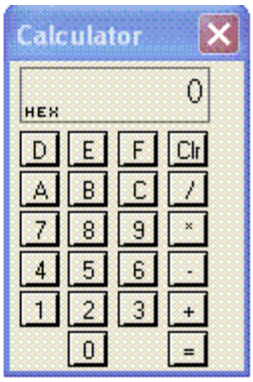


Информацию по сборке программатора PonyProg Вы найдете в "Приложении №1".

Следует учесть, что программатор ***PonyProg*** лучше всего работает на относительно "медленных" компьютерах старых выпусков, так как, в свое время, программа ***PonyProg*** создавалась под них.
При подключении программатора к современным быстродействующим компьютерам с "навороченными" операционными системами, могут возникнуть конфликты типа "нестыковки" программы ***PonyProg*** с операционной системой или превышения предельно допустимой скорости обмена данными между компьютером и программируемым ПИКом, то есть программатор может просто не заработать.

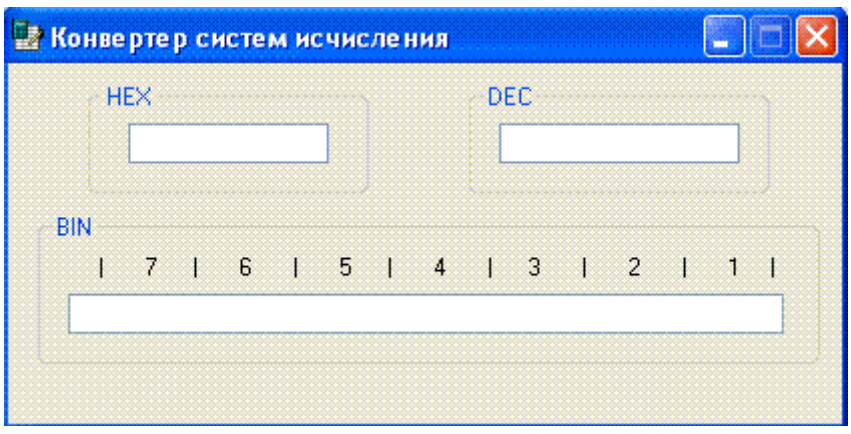
Это вовсе не есть факт, но такое может быть.
Самое лучшее решение - применение для этих целей компьютера с тактовой частотой до 500мГц и операционной системы ***Windows95/98***.
Лично я сделал так: купил практически "за так" "древнюю старушку", обманул BIOS, пристегнув к родному винчестеру, "помощником", дополнительный винчестер на 8Гб, поставил ***Windows98*** и включил "форсаж".
Получилось "дешево и сердито", и прежде всего по той причине, что при занятии программированием, создании печатных плат, вычерчивании схем и прочих радиолюбительских делах, особой скорости и не требуется, так как все эти занятия неспешны, и особого смысла задействовать под это дело быстродействующие компьютеры нет.
В моей "старушке" стоит почти на 2Гб подобного рода программ, в том числе и довольно-таки "навороченных", и ничего, прекрасно работает.
С тем, что должно быть в наличии обязательно, я надеюсь, понятно, а теперь о полезных "мелочах". Скачайте эти две маленькие, но полезные и удобные программки:

HEX - калькулятор: файл CALC32.rar прилагается (папка ***"Программы"*****).**
Конвертер систем исчисления: файл BCONV32 прилагается (папка ***"Программы"*****).**



Они настолько просты, что Вы без труда разберетесь, зачем они нужны.
Чаще всего требуется конвертер систем исчисления.

08.07.07.
Один из активных участников работы, **Петр Высочанский**, разработал программу конвертера систем исчисления, которая наиболее адаптирована к практическим нуждам:



Конвертер систем исчисления Петра Высочанского: файл Hex-Dec_Bin.exe прилагается (папка ***"Программы"*****)**

При открытии, программа конвертера устанавливает английскую раскладку клавиатуры (то, что нужно).

Итак, все что необходимо для занятия программированием PIC контроллеров имеется. Пока, это не более чем красивые и интригующие "штучки" не вполне понятного предназначения.
Можно их на досуге рассмотреть, на что-нибудь понажимать, только, во избежание дальнейших недоразумений, не трогайте настроек по умолчанию.
Переходите к следующему разделу.

2. Что такое микроконтроллер, и как он работает

Прежде всего, микроконтроллер это процессор со всеми его "атрибутами", плюс встроенная, энергонезависимая память (программ и данных), что позволяет отказаться от внешней памяти программ и поместить программу в его энергонезависимую память.
Это позволяет создавать очень простые (в схемотехническом отношении) и компактные устройства, выполняющие, тем не менее, достаточно сложные функции.
Иногда даже диву даешься: эта маленькая "штучка" заменяет целую "груду старого железа" (К555 и т.д.).
Любой микроконтроллер, по своим возможностям, конечно же, уступает процессору компьютера, но тем не менее, существует весьма обширный класс устройств, которые преимущественно реализуются именно на микроконтроллерах.
И в самом деле, компьютер в карман не положишь и от батареек его не запитаешь.
Поэтому, во многих случаях, микроконтроллерам просто нет альтернативы.
"Сердцем" микроконтроллера является арифметико - логическое устройство (АЛУ).
Проще всего его представить в виде банального калькулятора, кнопками которого управляет программа, написанная на языке ассемблер (то есть, программист).
Если вдуматься, то ничего особо сложного, в механизме управления такого рода калькулятором, нет.
И в самом деле, если нужно, например, сложить числа A и B, то в тексте программы сначала задаются константы A и B, а затем дается команда "сложить".
Программисту вовсе не обязательно знать, что происходит с нулями и единицами (разве только только для общего развития), ведь калькулятор он на то и калькулятор, чтобы избавить пользователя от "возни" с машинными кодами и прочими "неудобоваримостями".
Когда Вы работаете с компьютером, Вам и не нужно детально знать, что происходит в дебрях операционной системы.
Если Вы туда "полезете", то "с ума сойдете", а микроконтроллер, по своей сути, есть тот же самый компьютер, но только простой.
Программисту только нужно детально знать, каким именно образом "приказать железяке" сделать то, что необходимо для достижения задуманного.
Микроконтроллер можно представить себе как некий универсальный "набор" многофункциональных модулей (блоков), "рычаги управления" которыми находятся в руках программиста.
Этих "рычагов" достаточно большое количество, и естественно, их нужно освоить и точно знать, что именно произойдет, если "дернуть" (дать команду на языке ассемблер) за тот или иной "рычаг".
Вот здесь-то уже нужно знать, как "отче наше", каждую деталь и не жалеть на это "узнавание" времени.
Только таким образом пустую "болванку" (незапрограммированый ПИК) можно "заставить" выполнять какие-то "осмысленные" действия, результат большей части которых можно проверить в **симуляторе MPLAB** (об этом - позднее), даже не записывая программу в ПИК.
Итак, необходим переход к "модульному" мышлению.
Любой микроконтроллер можно уподобить детскому конструктору, в состав которого входит множество всяких предметов, манипулируя с которыми, можно получить тот или иной конечный "продукт".
Давайте с ними разберемся и "разложим все по полочкам".
В качестве примера я буду использовать один из самых распространенных PIC контроллеров **PIC16F84A**.
Он является как бы "проматерью" более сложных ПИКов, содержит минимальный "набор" модулей и как нельзя лучше подходит для первичного "въезда в м/контроллеры".

Энергонезависимая память.

Начнем с энергонезависимой памяти (память программ и память данных).
Информация, заложенная в энергонезависимую память, сохраняется при выключении питания, и поэтому именно в нее записывается программа.
То "место" энергонезависимой памяти, куда записывается программа, называется памятью программ.
Объем памяти программ может быть различен. Для PIC16F84A, он составляет 1024 слова.
Это означает, что он предназначен для работы с программами, объем которых не превышает

1024 слов.
Слово памяти программ не равно одному байту (8 бит), а больше его (14 бит).
Отдельная команда, которую ПИК будет в дальнейшем выполнять, занимает одно слово в памяти программ.
В зависимости от названия этой команды в ассемблере, слово принимает то или иное числовое значение в машинном коде.
После записи в ПИК "прошивки" программы, слова памяти программ (машинные коды) как бы "превращаются" в команды, которые располагаются, в памяти программ, в том же порядке, в котором они следуют в исходном тексте программы, написанном на языке ассемблер, и в том же порядке им присваиваются адреса, при обращении к которым, та или иная команда "извлекается" из памяти программ для ее выполнения.
Последовательность же их выполнения определяется логикой программы.
Это означает то, что выполнение команд может происходить не в порядке последовательного возрастания их адресов, с шагом в одну позицию (так называемый **инкремент**), а "скачком".
Дело в том, что только уж самые простейшие программы, в пределах одного их полного цикла, обходятся без этих "скачков", называемых **переходами**, и выполняются строго последовательно.
В остальных же случаях, так называемая (мной) "рабочая точка программы" "мечется по тексту программы как угорелая" (как раз благодаря этим самым переходам).
Термин **"рабочая точка программы"** - моя "самодеятельность".
В свое время, я был очень сильно удивлен отсутствием чего-то подобного в информации, связанной с объяснением работы программ.
Казалось бы, чего проще, по аналогии, например, с рабочей точкой транзистора, сделать более комфортным "въезд в механику" работы программ?
Так нет же, как будто специально, придумываются такие "головокружительные заменители", причем, в различных случаях, разные, что запутаться в этом очень просто.
Итак, рабочую точку программы можно представить себе в виде некоего "шарика от пинг-понга", который "скачет" по командам текста программы в соответствии с алгоритмом (логикой) исполнения программы.
На какую команду "шарик скакнул", та команда и исполняется.
После этого он "перескакивает" на другую команду, она исполняется, и т.д.
Эти "скачки" происходят непрерывно и в течение всего времени включения питания устройства (исполнения программы).
Любая более-менее сложная программа разбивается на части, которые выполняют отдельные функции (своего рода программки в программе) и которые называются **подпрограммами.**
Атрибут любой подпрограммы - функциональная законченность производимых в ней действий.
По сути своей, эта "выдумка" введена в программирование для удобства реализации принципа "разделяй и властвуй": "врага" ведь гораздо легче "разгромить по частям, чем в общей массе".
Да и порядка больше.
Безусловные переходы (переходы без условия) между подпрограммами (если они последовательно не переходят одна в другую), осуществляются при помощи **команд безусловных переходов**, в которых обязательно указывается адрес команды в памяти программ (косвенно - в виде названия подпрограммы или метки), на которую нужно перейти.
Существуют также переходы с условием (**условные переходы**), то есть, с задействованием так называемого **стека**.
Более подробно о переходах я расскажу позднее.
Адреса команд определяются счетчиком команд (он называется **PC**).
То есть, каждому состоянию счетчика команд соответствует одна из команд программы.
Если команда простая, то счетчик просто инкрементируется (последовательно выполняется следующая команда), а если команда сложная (например, команда перехода или возврата), то счетчик команд изменяет свое состояние "скачком", активируя соответствующую команду.
Примечание: **инкремент** ***- увеличение на единицу величины числа, с которым производится эта операция, а*** **декремент** ***- уменьшение на единицу*** (так называемые комплиментарные операции).
В простейшем случае, то есть в случае отсутствия в программе переходов, счетчик команд PC, начиная с команды "старта" (нулевой адрес), многократно инкрементируется,

последовательно активизируя все команды в памяти программ.
Это означает, что в большинстве случаев, за каждый так называемый **машинный цикл** (такт работы программы: для ПИКов он равен четырем периодам тактового генератора) работы ПИКа, происходит исполнение одной команды.
Есть и команды исполнение которых происходит за 2 машинных цикла (м.ц.), но их меньше. Команд, которые исполняются за 3 м.ц. и более нет.
Таким вот образом, на большинстве участков программы (я их называю **"линейными участками"**), последовательно и перебираются адреса в памяти программ (команды последовательно исполняются).
В более сложных программах, с большим количеством условных и безусловных переходов, работу счетчика команд PC можно охарактеризовать фразой "Фигаро здесь, Фигаро там".
1 машинный цикл (м.ц.) равен 4-м периодам тактового генератора ПИКа.
Следовательно, при использовании кварца на 4 Мгц., 1 м.ц.=1 мкс.
Выполнение программы, в рабочем режиме (кроме работы в режиме пониженного энергопотребления SLEEP), никогда не останавливается, то есть, за каждый машинный цикл (или за 2, если команда исполняется за 2 м.ц.) должно выполняться какое-либо действие (команда).
Тактовый генератор, формирующий машинные циклы, работает постоянно.
Если его работу прервать, то исполнение программы прекратится.
Может сложиться ложное представление о том, что работу программы можно на какое-то время остановить, используя одну или несколько команд – "пустышек", не производящих полезных действий (есть такая команда **NOP**).
Это представление не верно, так как в этом случае, речь идет только о задержке выполнения следующих команд, а не об остановке исполнения программы.
Программа исполняется и в этом случае, так как "пустышка" есть та же самая команда программы, только не производящая никаких действий (короткая задержка).
Если же нужно задержать выполнение каких-либо последующих команд на относительно длительное время, то применяются специальные, циклические подпрограммы задержек, о которых я расскажу позднее.
Даже тогда, когда программа "зависает" ("глюк"), она исполняется, просто только не так, как нужно.
Остановить (в буквальном смысле этого слова) исполнение программы можно только прекратив работу тактового генератора.
Это происходит при переходе в **режим пониженного энергопотребления (SLEEP)**, который используется в работе достаточно специфических устройств.
Например, пультов дистанционного управления (и т.д.).
Отсюда следует вывод: программы, не использующие режим **SLEEP** (а таких - большинство), для обеспечения непрерывного выполнения команд программы, обязательно должны быть циклическими, то есть, иметь так называемый **полный цикл программы**, причем, многократно повторяющийся в течение всего времени включения питания.
Проще говоря, рабочая точка программы должна непрерывно (не останавливаясь) "мотать кольца" полного цикла программы (непрерывно переходить с одного "кольца" на другое).

Общие выводы:

1. ***Команды программы "лежат" в памяти программ в порядке расположения команд в тексте программы.***
2. ***Адреса этих команд находятся в счетчике команд PC и каждому адресу соответствует одна из команд программы.***
3. ***Команда активируется (исполняется), если в счетчике команд находится ее адрес.***
4. ***Активация команд происходит либо последовательно*** (на "линейном" участке программы***), либо с переходом ("скачком") на другую команду*** (при выполнении команд переходов), ***с которой может начинаться как подпрограмма*** (переход на исполнение подпрограммы), ***так и группа команд, выделенная меткой*** (переход на исполнение группы команд, которой не присвоен "статус" подпрограммы).
5. ***Выполнение команд программы никогда не останавливается*** (за исключением режима SLEEP)***, и поэтому программа должна быть циклической.***

Если сейчас это не понятно, то ничего страшного. "Просветление" будет позже.

Кроме памяти программ, PIC16F84A имеет энергонезависимую память данных (EEPROM память данных).
Она предназначена для сохранения данных, имеющих место быть на момент выключения питания устройства, в целях их использования в дальнейшем (после следующего включения питания).
Так же, как и память программ, память данных состоит из ячеек, в которых "лежат" слова.
Слово памяти данных равно одному байту (8 бит).
В PIC16F84A, объем памяти данных составляет 64 байта.
Байты, хранящиеся в памяти данных, предназначены для их считывания в стандартные 8-битные регистры, речь о которых пойдет далее.
Данные из этих регистров могут быть записаны в **EEPROM память данных**, то есть, **может быть организован обмен данными между памятью данных и регистрами.**
Например, именно EEPROM память данных я использовал в своем частотомере для сохранения последних, перед выключением питания, настроек.
Она же используется и для установки значений промежуточной частоты.
Во многих программах, память данных вообще не используется, но это "вещь" исключительно полезная, и далее я расскажу о ней подробнее.

Регистры

Включите свое образное мышление.
Предположим, что Вы купили механический конструктор.
Что Вы делаете сначала?
Изучаете его составные части для того, чтобы прикинуть, куда что можно приспособить, что как можно соединить друг с другом, какие из нескольких отверстий нужно выбрать, чтобы соединить несколько деталей и т. д.
Если Вы хотите собрать из него нечто, то Вы, используя результат полученных знаний, вырабатываете определенный план и реализуете его.
По сути, то же самое происходит и при работе с ПИКами.
"План" это программа, а "составные части конструктора" есть регистры.
Производя манипуляции с регистрами и встраивая их в программу (в замысел), можно реализовать огромное количество различных устройств.
Без детального знания всех элементов этого "ПИК – конструктора", не может быть и речи о какой-либо эффективной работе по составлению программ. Уделите этому особое внимание.
В ПИКах существуют две группы регистров:
- **регистры общего назначения (GRP)**
- **и регистры специального назначения (SFR).**

Эти регистры являются элементами оперативной памяти.
То есть, они сохраняют информацию только при включенном питании.
Все регистры как общего, так и специального назначения, по своему "объему", являются однобайтными.
Это - общий стандарт.

Регистры общего назначения

Это "вотчина" программиста.
Изначально, они похожи на "пустые болванки" и толка от них - никакого.
Чтобы этот толк был, необходимо "прописать" регистр в программе ("ввести его в эксплуатацию"), и в случаях обращений к содержимому этого регистра, произвести, с этим содержимым, замысленные программистом действия (то же самое относится и к регистрам специального назначения).
Регистры общего назначения используются как однобайтная (или многобайтная, если для этой цели используются несколько регистров) оперативная память.
В основном, они используются для того, чтобы сохранить какие-либо числа, с целью использования их в дальнейшем.
Причем, по ходу исполнения программы, эти числа могут изменяться: увеличиваться, уменьшаться, сбрасываться, снова загружаться (как в неизменном, так и в измененном виде) и т.д..
В ходе выполнения программы, один и тот же регистр общего назначения (или группа этих регистров) может использоваться для работы не только с одной, но и с двумя или более функционально независимыми группами числовых данных.

Например, в одной подпрограмме он может работать как счетчик, а в другой - как буферная память, но на первых порах, этим пока "не стоит забивать голову", так как количество регистров общего назначения достаточно велико ("дефицита" в них нет).
Когда, например, говорят, что на регистрах общего назначения **X, Y, Z** (их можно назвать как угодно, это дело вкуса программиста) собран 3-х разрядный вычитающий счетчик, то следует иметь ввиду, что эти регистры, сами по себе, таковым счетчиком не являются.
Этим счетчиком их делает программа, а если конкретнее, то программист.
Именно программа (соответствующие ее команды и последовательность их исполнения) определяет порядок взаимодействия разрядов счетчика (старший, средний, младший), направление счета, точку начала и конца отсчета, наличие или отсутствие предварительной установки, момент сброса и т.д.

Откройте *"Приложение №2".*

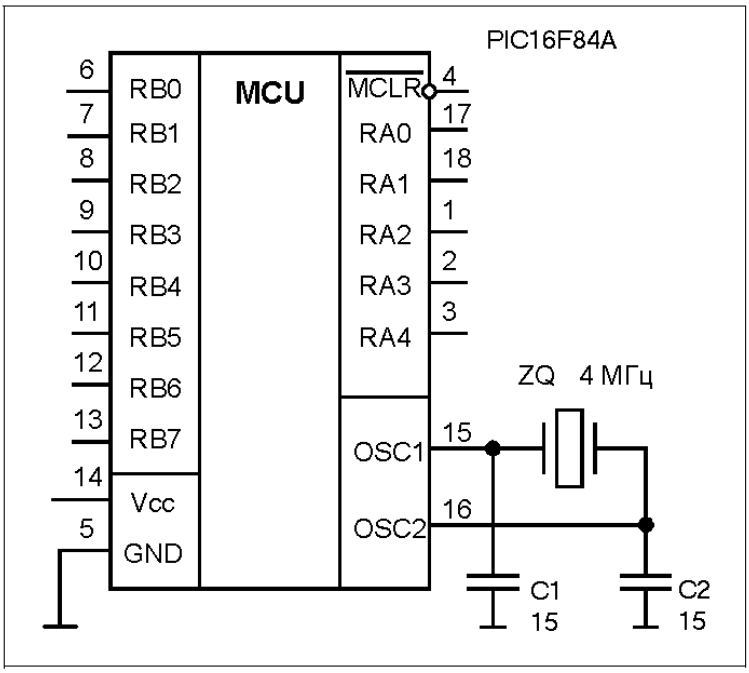
На этой картинке Вы видите область оперативной памяти PIC16F84A.
Регистры общего назначения находятся в пустых клеточках.
Каждой такой клеточке соответствует свой адрес.
Пустые (без названий) они потому, что названия им назначает программист.
Например, для работы в программе, необходим регистр буферной памяти, в котором нужно "попридержать" число, с целью его использования в дальнейшем.
Назовем его, например, Mem и присвоим ему адрес **0Ch** (это делается в "шапке" программы, о чем - позже).
Это означает то, что регистр Mem займет в области оперативной памяти ту клеточку, которая расположена правее желтой клеточки с надписью INTCON.
Всё. Регистр буферной памяти условно "прописан" (ему назначено название и присвоен адрес в области оперативной памяти) и теперь с ним можно работать.
"Условно" потому, что пока он "прописан" на бумаге, а не в тексте программы
(об этой "технологии" – позже).
Эту картинку я делал для себя, и кроме обучения, она полезна в работе.
Распечатайте ее в нескольких экземплярах.
При составлении программы, в пустые клетки этой таблицы, можно заносить названия регистров общего назначения, а при необходимости, и какие-либо пояснительные надписи.
Это удобно, и лично я, частенько так делаю.
Я надеюсь, что Вы без труда поймете, как определяется адрес той или иной пустой клетки (да и закрашенной тоже).
В конце адреса стоит буква **h** (признак 16-ричной системы исчисления).
Именно в таком виде лучше всего указывать адрес регистра в "шапке" программы.
Например, **0Ch**, **14h**, **29h**, **2Ch** ...
Можно использовать и прописные буквы: **0ch**, **2dh**, **1fh** ...
Хотя в распечатке области оперативной памяти указаны 36 регистров общего назначения, но на самом деле их **68** (еще плюс 2 ряда по 16 штук ниже нижнего ряда).
Таким образом, 68 регистров общего назначения, в области оперативной памяти, имеют адреса: **банк 0** - с **0Ch** по **4Fh**, **банк 1** - с **8Ch** по **CFh**.
При составлении программ, редко когда возникает необходимость в задействовании всех 68-ми регистров общего назначения, и поэтому, в распечатке области оперативной памяти, я указал не 68, а 36 регистров общего назначения, чего вполне достаточно для составления даже достаточно сложных программ. Имейте это ввиду.
Область оперативной памяти, "во всей своей полномасштабной красе", можно наблюдать в MPLAB, речь о которой пойдет ниже.
Область оперативной памяти разделена на 2 так называемых **банка** (в таблице их разделяет черная, жирная линия).
Банк он и есть банк, то есть, некое "хранилище задействованных или незадействованных активов" (в виде регистров общего и специального назначения).
Сверху расположен самый "ходовой", **нулевой банк**, в котором программы работают бОльшую часть своего времени.
Что касается регистров общего назначения, то в части касающейся PIC16F84A, все они расположены в нулевом банке, а в первом банке просто дублируются (повторяются или, как часто говорят, отображаются).
Такого рода дублирование позволяет работать с регистрами общего назначения как в нулевом, так и в первом банке, "не забивая себе голову" переходами из банка в банк, чего не

скажешь о недублирующихся регистрах специального назначения.

Регистры специального назначения

Детали нашего ПИК-конструктора состоят не из одних регистров общего назначения.
Это - самые простые "детали".
По большому счету, к его "деталям" относится практически вся электронная "начинка" PIC контроллера: тактовый генератор, система сброса, АЛУ, память, устройства, обеспечивающие прерывания, порты ввода - вывода, таймер TMR0, предделитель, сторожевой таймер WDT (перечислен минимальный "набор" для PIC16F84A. Для более сложных ПИКов этот набор еще больше).
Перечисленные выше устройства имеют по нескольку режимов работы, конкретный режим работы которых выбирает программист.
Этот выбор режимов происходит в регистрах специального назначения, путем установки соответствующих их битов в 0 или 1, а также путем установки в 0 или 1 соответствующих битов в так называемом **слове конфигурации**.
Что касается последнего, то на первых порах, для простоты понимания, можно условно считать, что биты конфигурации "лежат" в дополнительном регистре "суперспециального" назначения, который не отображается в области оперативной памяти.
Не отображается потому, что биты конфигурации расположены в специальном "секторе" энергонезависимой памяти микроконтроллера.
В дальнейшем, такое упрощение позволит, без особого "напряга", представить себе механизм работы с битами конфигурации.
Рассмотрим **регистры специального назначения (SFR).**
Взгляните в область оперативной памяти.
Эти регистры находятся в закрашенных желтым цветом клеточках (адреса 07h и 87h – "пустышки"), и в них же находятся их названия.
Названия и адреса этих регистров стандартны (неизменны) и манипулировать ими (назначать свои), в отличие от регистров общего назначения, нельзя.
А раз это так, то единственное, что можно с ними сделать, так это только изменить их содержимое, то есть, произвести те или иные манипуляции с их битами (установить те или иные из них их в 0 или 1).
В зависимости от этих манипуляций, электронная "начинка" ПИКа приобретает свойства, обеспечивающие функционирование программы (а значит и устройства) по замыслу программиста.
Регистры SFR с названиями, выделенными **красным цветом**, дублируются (отображаются) в обеих банках.
Черным цветом выделены названия регистров SFR, которые не дублируются.
Работа с большей частью регистров SFR происходит в нулевом банке.
При этом, содержимое регистров SFR нулевого банка, выделенных **красным цветом**, дублируется (отображается) в первом банке.
Для работы с регистрами SFR 1-го банка, выделенными **черным цветом**, необходимо (при обращении к любому из них) перейти в 1-й банк, произвести с их содержимым необходимые действия, и вернуться после этого назад, в нулевой банк.
В нулевом банке, действий с их содержимым производить нельзя.
Таких регистров 5 (см. область оперативной памяти), и их названия нужно хорошо запомнить именно по причине необходимости смены банка.
Во всех остальных случаях работа происходит в нулевом банке.
Некоторые регистры специального назначения, кроме битов, непосредственно влияющих на настройки "начинки" ПИКа, имеют еще и биты, которые на эти настройки не влияют.
Их называют **флагами**.
Проще говоря, эти биты не производят действий, непосредственно приводящих к каким-либо изменениям, но обращаясь к их содержимому, можно считать информацию о результате какой-либо операции (например, является ли результатом этой операции ноль или нет), а затем использовать эту информацию при выборе одного из нескольких сценариев дальнейшей работы программы.
Таким образом, флаги могут опосредованно (косвенно) участвовать в рабочих действиях и вовсе не являются чем-то бесполезным.
При отладке программы, проанализировав состояния соответствующих флагов, можно узнать много полезной информации.

В свое время, я долго и нудно перерабатывал информацию, из различных источников, в поисках наиболее наглядного и понятного варианта распечатки состава регистров SFR, после чего плюнул на это дело и самостоятельно составил эту распечатку для себя.
Мне она сослужила "хорошую службу", пусть послужит и Вам.
Совет: изначально, Вам будет сложновато запомнить то, что Вы увидите.
Поэтому сделайте так, как сделал я: на Вашем рабочем месте, прикрепите эти листы так, чтобы они висели перед Вами и постоянно бросались Вам в глаза.
Общая информация по SFR: ***"Приложение №3".***
Регистр OPTION: ***"Приложение №4".***
Регистр STATUS: ***"Приложение №5".***
Регистр INTCON: ***"Приложение №6".***
Регистр EECON1: ***"Приложение №7".***
Биты конфигурации : ***"Приложение №8".***



Все эти файлы относятся к PIC16F84A.
Таблица перевода чисел из одной системы исчисления в другую: ***"Приложение №9".***
Это - одна из наиболее востребованных таблиц, и при работе с программой, Вы будете часто к ней обращаться.
В ней расписан "расклад" чисел в пределах одного байта.
В большинстве случаев, этого достаточно, но если Вы будете работать с числами, "находящимися за пределами" одного байта, то используйте конвертор систем исчисления.
Буквами **D, B, H** обозначены **десятичная, бинарная, 16-ричная системы исчисления** соответственно.
Небольшое "лирическое отступление" для поднятия боевого духа: если, после прочтения изложенной выше информации, Вы начнете ощущать, что ожидаемое понимание не приходит, то особо не расстраивайтесь.
На данной стадии "въезда", "бардак в мозгах" нормален и естественен.
Я очень удивлюсь, если этого не будет.
В такого рода информации нужно немного "повариться", что предполагает приложение некоторых усилий.
"Кавалерийским наскоком" такие дела не делаются.
Сейчас "выстраивается скелет" и нужно отдельно разобраться с каждой "косточкой".
Дело это конечно "муторное", но совершенно необходимое, а "прозрение" придет позднее.
Желаю Вам терпения, усидчивости и хорошей злости.
Пошли дальше.
Разбираемся с битами регистров SFR.
Сразу расставлю приоритеты: в первую очередь, с целью недопущения "бардака", сначала я буду рассказывать о рабочих битах, а о флагах расскажу отдельно.
С этой же целью, на первых порах, я не буду вдаваться в некоторые необязательные, для начинающих, детали (о них - позднее).
Сразу обращаю Ваше внимание: ***нумерация битов в байте происходит не слева - направо, а справа - налево.***
Это означает то, что бит младшего разряда (бит с номером ноль) - крайний справа, а не слева.
7-бит - самый старший. Про нумерацию битов слева направо - забудьте.
В распечатках, **красным цветом** обозначены рабочие биты, **зеленым цветом** - флаги, **серым цветом** – "пустышки" (не используются).

В регистре STATUS расположены три рабочих бита (**5,6,7**), из которых нас, пока, интересует только **5**-й бит с названием **RP0**.
Именно этим битом **переключаются банки** (**0 - нулевой банк, 1 - первый банк**).
На момент старта (начала исполнения) программы, в ПИКе всегда автоматически (по умолчанию) выставляется нулевой банк.
Бит **RP1** устанавливается равным нулю и для смены банка достаточно изменять только значение бита **RP0** (по умолчанию **RP0** устанавливается в **0**).
Учитывая то, что **7**-й бит (**IRP**) также, по умолчанию, устанавливается в **0**, и в PIC16F84A 2-го и 3-го банка просто нет (они есть в более сложных ПИКах), то про биты **6** и **7** можно просто "забыть" (конечно, условно, до поры до времени) и пока не "забивать себе ими голову".
Итак: в регистре **STATUS**, в приложении к PIC16F84A, имеется всего один рабочий бит переключения банков **RP0**.
Регистр OPTION.
Все 8 битов - рабочие.
Биты **PS0,PS1,PS2** определяют коэффициент деления предделителя (см. таблицу в распечатке).
Предделитель это последовательная цепочка из 8-ми триггеров, каждый из которых делит на 2.
Таким образом, максимальный коэффициент деления предделителя = 256, и он, в пределах разрешенных таблицей значений, может задаваться значениями битов **PS0,PS1,PS2**.
Предделитель может быть включен либо **перед** таймером **TMR0**, либо **после** сторожевого таймера **WDT**.
Это определяет **3**-й бит регистра OPTION (**PSA**).
Таймер **TMR0** - то же самое, что и предделитель (Кделения = 256), но с возможностью предустановки (предварительной установки) и синхронизации.
Предустановка это есть запись в таймер числа, начиная с которого происходит счет.
Обычно, такого рода загрузку (числовую коррекцию) производят для задания некой начальной точки отсчета.
Если таймер **TMR0** работает без предустановки, его коэффициент деления всегда равен 256 и не меняется.
Достаточно специфическим случаем изменения Кделения **TMR0** является случай работы с периодическими предустановками. Это применяется относительно редко.
В большинстве же случаев, перед **TMR0** включается предделитель с заданным Кделения.
Если предделитель включен перед **TMR0**, то, при поступлении команды сброса (уст. 0), они сбрасываются одновременно.
Команду сброса можно рассматривать как разновидность предустановки.
Основная функция таймера **TMR0** - подсчет количества импульсов за определенное программой время.
Через каждые 256 импульсов (при переходе из состояния FFh в состояние 00h) происходит так называемое переполнение таймера (переход на новое кольцо счета), количество которых (при необходимости подсчета количества импульсов большего, чем 256) подсчитывается.
Например: за 1сек. произошло 10 переполнений, значит **TMR0** посчитал 2560 импульсов.
Если приплюсовать к этому количеству содержимое **TMR0** на момент окончания счета, то получим точное количество импульсов, поступивших на вход **TMR0** за 1сек.
Если перед **TMR0** включен предделитель, то в итог подсчета вносятся соответствующие коррективы, определяемые заданным коэффициентом деления предделителя и числом, "лежащим" в предделителе на момент окончания счета.
На такого рода подсчете и основывается принцип работы устройств, производящих подсчет импульсов за заданный интервал времени.
Сторожевой таймер WDT это RC-одновибратор (RC-ждущий мультивибратор) с перезапуском, формирующий импульс длительностью примерно **18мс.**
Если работа **WDT** разрешена (а она может быть и запрещена), то после старта программы, он запускается, и если его, в интервале времени 18мс., не перезапустить, то он окончит формирование импульса, и по его заднему фронту, сформируется сигнал сброса, после чего программа начнет исполняться со своего начала.
Зачем это нужно?
В случае "зависания" программы, **WDT**, сбросив программу на начало, может вывести ее из этого нехорошего состояния.
Именно поэтому таймер **WDT** и назван **сторожевым**.

Он как бы "сторожит глюк", и как только он происходит (при этом, сброса **WDT** не происходит), "говорит свое веское слово".
Для обеспечения этого сторожевого режима, в ходе выполнения программы, необходимо периодически (через время не более 18мс.) сбрасывать **WDT** (не допускать его срабатывания).
Если после **WDT** поставить предделитель, то период сброса **WDT** можно увеличить (это зависит от заданного Кделения предделителя).
3-й бит (**RSA**) регистра **OPTION** определяет, к чему подключить предделитель (к **TMR0** или **WDT**).
Если предделитель включен после **WDT**, то они оба сбрасываются по команде сброса **WDT**.
4-м битом регистра **OPTION** (**TOSE**) устанавливается момент срабатывания таймера **TMR0**.
При установке бита **TOSE** в **1**, счет происходит по спадам импульсной последовательности (переход от 1 к 0), присутствующей на выводе **RA4/TOCKI**, а при установке его в **0** - по фронтам (переход от 0 к 1).
Значение **5**-го бита регистра **OPTION** (**TOCS**) определяет, какой сигнал будет подаваться на вход **TMR0**: либо внешний сигнал со счетного входа **RA4/TOCKI** (бит устанавливается в **1**), либо внутренний тактовый сигнал **CLKOUT** (бит устанавливается в **0**).
В качестве внутреннего тактового сигнала используется сигнал с частотой опорного генератора ПИКа, разделенной на 4, то есть, в этом случае, **TMR0** будет считать каждый машинный цикл (1мкс. при применении кварца на 4мГц).
Это - для случая работы **TMR0** без предделителя.
Если используется предделитель, то нужно учесть его коэффициент деления.
6-й бит регистра **OPTION** (**INTEDG**) определяет, по какому именно перепаду, на входе внешнего прерывания **INT**, будет начинаться выполнение подпрограммы прерывания (о прерываниях - позднее).
1 – "уход" в подпрограмму прерывания будет происходить по фронту сигнала на выводе **RB0/INT**, **0** - по спаду.
В зависимости от состояния **7**-го бита регистра **OPTION** (**-RBPU**), к выводам порта B либо подключаются, либо не подключаются подтягивающие резисторы (между выводами порта B и плюсом источника питания).
В PIC16F84A, они подключаются или отключаются "оптом", то есть, все 8, а не выборочно.
Обращаю Ваше внимание на то, что **подтягивающие резисторы могут быт подключены, к выводам порта B, только в случае установки режимов их работы "на вход"**.
Поясняю: выводы порта B могут работать как "на вход", так и "на выход".
При работе "на выход", подтягивающие резисторы автоматически отключаются (если они были подключены), так как выходы защелок портов имеют свои подтягивающие резисторы, включенные постоянно (нагрузки защелок).
При работе "на вход" подтягивающие резисторы, если они были подключены, не отключаются.
Таким образом, подтягивающие резисторы, если они подключены, могут быть нагрузкой выходных каскадов внешних устройств, подключенных к выводам порта B, работающим "на вход".
Это удобно в случаях отсутствия, в этих каскадах, "своей" нагрузки.
То есть, подтягивающие резисторы необходимо включить, если к выводу порта B, работающему "на вход", подключается выход внешнего (по отношению к ПИКу) устройства с открытым коллектором или открытым стоком.
В этом случае, соответствующий подтягивающий резистор порта B будет являться нагрузкой внешнего устройства с открытым коллектором или открытым стоком.
Если выходные каскады внешних устройств имеют "свою" нагрузку, то подтягивающие резисторы подключать не нужно.
Иначе, не во всех, но в некоторых случаях, это может привести к изменению режима работы выходного каскада внешнего устройства по постоянному току.
Регистр **EECON1** управляет чтением - записью в EEPROM память данных.
Он имеет 3 рабочих бита, манипуляции со значениями которых необходимы при организации процедур чтения - записи.
Пока, не нужно "забивать себе голову" информацией об этих битах.
Причина простая: при чтении и записи данных в EEPROM память данных, применяются рекомендуемые разработчиками стандартные процедуры (эти биты в них участвуют) и вносить в них коррективы просто нет необходимости.

Если нужно произвести чтение или запись, то в программу просто вставляется соответствующая стандартная процедура, рекомендованная разработчиками, в которой все действия стандартно "расписаны" (включая и работу с этими битами).
Об этих процедурах я расскажу позднее.
Сказанное выше относится и к регистру **EECON2**.
Регистр **INTCON** имеет 5 рабочих битов.
Это регистр управления прерываниями.
После возникновения факта прерывания, за счет "ухода" рабочей точки программы в подпрограмму прерываний, происходит временная приостановка выполнения "основной" (условно) программы.
После возврата из подпрограммы прерываний, отработка "основной" программы продолжается.
Если в составе программы имеется подпрограмма прерываний, то в "шапке" программы (пока привыкните к этому названию, а объяснение будет позднее) устанавливается так называемый вектор прерываний, который также, как и команда старта для "основной" программы, определяет начальный адрес счетчика команд для первой команды, но только не "основной" программы, а подпрограммы прерываний.
"Основная" программа стартует и выполняется до тех пор, пока рабочая точка программы не войдет в "зону" разрешения прерываний.
Если в этой "зоне" происходит, например, внешнее прерывание по входу **RB0/INT**, то адрес следующей команды "основной" программы запоминается (адрес возврата записывается в стек - об этом позднее), выполнение "основной" программы приостанавливается, и происходит переход (по вектору прерывания) на начало исполнения подпрограммы прерывания.
Далее, подпрограмма прерываний выполняется по такому же принципу, как и "основная" программа, вплоть до окончания этой подпрограммы.
В конце подпрограммы прерываний всегда исполняется специальная команда возврата.
При этом, из стека извлекается адрес возврата, после чего "основная" программа начинает исполняться далее.
Таким образом, прерывание - это специфический переход, в большинстве случаев, по внешнему воздействию, из "основной" программы, на выполнение подпрограммы прерывания, с последующим возвратом из не в "основную" программу.
В ПИКах имеется несколько источников прерываний.
Для того, чтобы произошло любое из них, сначала нужно установить бит глобального разрешения прерываний **GIE** (установить в 7-м бите регистра **INTCON** единицу).
6-й бит (**EEIE**) разрешает/запрещает прерывания по окончанию цикла записи в EEPROM память данных.
5-й бит (**TOIE**) разрешает/запрещает прерывания по переполнению TMR0, то есть, если этот вид прерываний разрешен, то уход в подпрограмму прерывания произойдет при смене содержимого TMR0 с FFh на 00h (при переполнении).
4-й бит (**INTE**) разрешает/запрещает внешнее прерывание по входу RB0/INT.
3-й бит (**RBIE**) разрешает/запрещает прерывания по изменению уровня сигнала на любом из выводов RB4…RB7.
Примечание: направление работы выводов портов A и B можно устанавливать в любых комбинациях (см. регистры **PORTA** и **PORTB**. О них - ниже).
Те выводы портов, на которые поступают сигналы, инициирующие прерывания, должны работать "на вход".
Это относится к выводам **RB0/INT**, **RB4...RB7** порта B и выводу **RA4/TOCKI** порта A.
Что касается последнего, то импульсную последовательность, подаваемую на счетный вход TMR0 (вывод RA4/TOCKI), вполне можно считать сигналом прерываний, только косвенно, так как прерывание наступает не по факту подсчета отдельного импульса, а по факту подсчета группы импульсов (переполнение TMR0).
Регистры **TRISA** и **TRISB** имеют 5 и 8 рабочих бита соответственно (по количеству выводов портов A и B). Их адреса - 85h и 86h (см. область оперативной памяти).
Распечатки на них нет, но Вы без труда поймете их функции.
Биты регистров **TRISA** и **TRISB** управляют направлением работы выводов портов A и B.
Если какой-нибудь из этих битов устанавливается в **1**, то соответствующий вывод порта работает **"на вход"**, то есть, принимает данные с выхода внешнего устройства, подключенного к этому выводу.

Если бит **TRISA/TRISB** устанавливается в **0**, то соответствующий вывод порта уже сам является источником сигнала для подключенного к этому выводу, входа внешнего устройства, и программа может управлять этим внешним устройством (например, светодиодом или каким-то другим исполнительным устройством).
При помощи этих битов, в пределах каждого из портов, можно устанавливать различные комбинации направлений работы их выводов, а также и менять их (направления) в ходе исполнения программы.
Так как регистры **TRISA** и **TRISB** находятся в 1-м банке, то, при работе с ними, необходимо перейти в 1-й банк, произвести необходимые изменения битов и вернуться в 0-й банк (если далее работа происходит в нулевом банке).
Регистры **PORTA** и **PORTB** (см. распечатку: ниже регистра **STATUS**) управляют защелками портов A и B.
Количество рабочих битов в них такое же, как и количество выводов портов (5 и 8).
Защелка это аппаратно реализованное устройство оперативной памяти, а проще говоря, триггер.
Биты регистров **PORTA** и **PORTB** управляют этими триггерами.
Если какой-либо из этих битов установить в **1**, то на выходе защелки (а значит и на соответствующем выводе порта, настроенном "на выход") также установится **1**
(а для ноля - ноль).
Так как защелки являются триггерами, то перевод их выходов из одного состояния в другое происходит "одномоментно".
Это означает то, что, например, для вывода цифры на 7-сегментный индикатор, в течение определенного программой интервала времени, достаточно одной команды на вывод цифры (байта) в начале этого интервала времени, и одной команды на ее сброс в конце этого интервала времени.
В промежутке времени между этими двумя командами, программа может выполнять какие-либо другие действия, формируя за счет этих действий данный интервал времени, причем эти действия могут быть напрямую не связанными с работой по выводу цифры в 7-сегментный индикатор (например, может производиться подсчет количества импульсов, операции с регистрами, работа с EEPROM памятью и т.д.).
Следует четко уяснить следующее: биты регистров **PORTA** и **PORTB** защелками управляют всегда, но не всегда данные с выходов защелок присутствуют на выводах портов.
Выходы защелок портов подключаются к выводам портов только при работе этих выводов "на выход".
При работе выводов портов на вход, выходы защелок от соответствующих выводов портов отключаются.
В отличие от выводов порта B, к которым могут быть либо подключены, либо отключены внутренние, подтягивающие резисторы, выводы порта A внутренней "подтяжки" не имеют.
Таким образом, если они настроены на работу на вход, то к ним нужно подключать внешние подтягивающие резисторы.
Если нужно организовать счет импульсов (задействован **TMR0**), то роль подтягивающего резистора для вывода **RA4/TOCKI** может выполнять, например, коллекторная нагрузка внешнего устройства.

Регистр счетчика команд **PC** (13 бит) разделен на 2 регистра.
В регистре счетчика команд PC с названием **PCL** (младший байт счетчика команд PC) находится младшие 8 битов адреса команды.
Остальные 5 битов (старшие) "лежат" в старшем байте счетчика команд PC с названием **PCH** (3 старших его бита не используются).
Изначально, "понятийновыгодно" представить себе 13-битное слово регистра PC как "неразделимый массив" с одним исключением (о нем - ниже), что (без учета этого исключения) означает: если команды отрабатываются последовательно (без "прыжков") и произошло переполнение **PCL** (состояние 255 сменилось на состояние 0), то содержимое **PCH** инкрементируется (увеличивается на 1).
13-разрядное слово позволяет организовать максимальный объем памяти программ до **256x2x2x2x2x2=8192** слов (в тексте программы можно использовать до 8192-х команд).
Для 5-битного **PCH**, это "потолок".
Могут использоваться не все 5 бит, а меньшее их количество.
Соответственно, объем памяти программ, в этих случаях, будет меньше.

Память программ многих типов м/контроллеров меньше 8192 слов, но есть и м/контроллеры с объемом памяти программ по максимуму (8 килослов).
С регистром **PCL** никаких проблем нет.
Проблемы есть с регистром **PCH**.
В частности, они связаны с вычисляемым переходом.
Вычисляемым переходом (это стандартное название процедуры) **называется приращение содержимого PC на величину числового содержимого регистра, называемого аккумулятором W** (об этом - в следующих разделах).
Проще говоря, если перед выполнением команды вычисляемого перехода, в PC имела место быть команда, например, с адресом 5 , а в регистре W "лежит" число 2, то, в результате исполнения команды вычисляемого перехода, произойдет переход ("прыжок") на команду с адресом 5+2=7.
Это упрощенный пример, который я привожу для обозначения сути вычисляемого перехода. На самом деле, существуют некоторые нюансы (суть от этого не меняется), о которых я расскажу позднее.
Таким образом, вычисляемый переход является четвертой разновидностью переходов по тексту программы после переходов по стеку (условный переход), переходов в подпрограммы (безусловный переход) и переходов на метки (разновидность безусловного перехода).
Какой именно из этих переходов применить - решает программист в соответствии с оптимальным, на его взгляд, способом решения стоящих перед ним задач.
Выше было сказано, что в том случае, если программа отрабатывается последовательно, при переполнении регистра **PCL**, происходит инкремент содержимого регистра **PCH**.
Это имеет место быть, но только не в случае исполнения процедуры вычисляемого перехода.
В этом случае, при переполнении **PCL**, инкремента **PCH** не происходит.
В технических описаниях м/контроллеров (речь идет о ПИКах среднего семейства) причина этого не объясняется, а только дается указание о недопустимости пересечения границы блока памяти программ (каждый блок 256 команд).
Поэтому можно только предположить, что это ограничение объясняется какими-то специфическими особенностями организации взаимодействия **PCL** и **PCH**, детали которого разработчики ПИКов не посчитали нужным разъяснить (просто поставили перед фактом).
И это не пустые слова.
Нарушение этого указания приводит к тому, что вычисляемый переход происходит не на ту команду, на которую нужно перейти (сбой работы программы).
Пример: начало исполнения вычисляемого перехода приходится на команду с номером, в памяти программ, равным 508.
508 - 256 = 252.
Число 252 вплотную, слева, "подступает к границе" 256-байтного блока (этой "границей" является переход от 255 к 0, и, в этом случае, нужно "держать ушки на макушке" в смысле: "а как бы ее не пересечь".
Если максимальное приращение счетчика команд PC будет менее числа 3 включительно, то "произойдет переход на планируемую программистом команду (509, 510, 511), а если оно будет более трех, например, 4, то, в результате сложения, содержимое регистра **PCL** окажется нулевым (252+4=256: 0 и 256 - одно и то же, но общепринято указывать 0), плюс инкремент содержимого регистра **PCH**.
Но, в случае вычисляемого перехода, инкремента содержимого регистра **PCH** не происходит.
В конечном итоге, переход на 512-ю команду не осуществляется, а вместо этого осуществляется возврат назад, на 256-ю команду (в **PCH** "лежало" число **00000001**, и оно, в случае вычисляемого перехода, не изменилось, а в регистре **PCL** будет "лежать" **00000000**), что есть сбой.
Пояснения к этому примеру:
Двоичное представление числа 508: **00000001** (**PCH**) **11111100** (**PCL**).
Для вычисляемого перехода с приращением 3, результат: **00000001 11111111** - все в норме: переход будет осуществлен на 508+3=511-ю команду.
Для вычисляемого перехода с приращением 4, должен быть результат **00000010 00000000**, но на самом деле, будет результат **000000001 00000000** (приращения **PCH** нет), то есть, 256.
Переход будет осуществлен на 256-ю команду, а должен быть на 512-ю команду (сбой).
Для того чтобы предотвратить подобного рода сбои, операцию вычисляемого перехода не нужно организовывать в непосредственной близости от "границ" 256-командных блоков памяти программ.

Практический вывод: в части касающейся PIC16F84A (в PC 1024 слова), если вычисляемого перехода нет, то о регистре **PCH** можно вообще "забыть", то есть, не обращать на него внимания, так как содержимое этого регистра будет изменяться "в автомате" и вмешательства программиста в этот процесс просто не требуется.
Если производится вычисляемый переход, то в ходе его исполнения, переполнения регистра PCL допускать нельзя.

Регистры **INDF** и **FSR** - регистры косвенной адресации.
Здесь необходимо кое-что пояснить.
Существует 2 вида адресации: **прямая** и **косвенная**.
Прямая адресация это обращение к регистру напрямую, то есть в команде указывается название регистра, с содержимым которого нужно произвести какое-то действие.
При этом, адрес этого регистра в области оперативной памяти, определяется в "автомате" (по его названию).
При косвенной адресации, название регистра не указывается, а указывается адрес этого регистра в области оперативной памяти, предварительно записанный в регистр **FSR**.
Для того, чтобы осуществить действие с регистром, адрес которого записан в регистре **FSR**, необходимо обратиться к регистру **INDF**.
Для начинающих это звучит достаточно непонятно.
Пока примите это к сведению.
Это нужно уяснять на конкретных примерах.
Конечный результат прямой и косвенной адресации один и тот же, просто в некоторых случаях, используя косвенную адресацию, можно обойтись гораздо меньшим количеством команд.
Сначала нужно как следует освоить прямую адресацию, как более простую для понимания, а только после этого переходить к косвенной.
И то, и другое предстоит сделать.
И наконец, самый часто используемый регистр - регистр **W** или его еще называют **аккумулятором**.
Это тот же регистр общего назначения, только с определенным названием (в том смысле, что оно заранее определено так же, как и названия регистров специального назначения).
Его адрес, в отличие от регистра общего назначения, знать не обязательно и "прописывать" его адрес не нужно.
Стоит только обратиться к регистру **W**, как адрес автоматически поставится ему в соответствие.
Он также является 8-битным и выполняет функцию оперативной памяти, обычно, на время одного, двух, трех машинных циклов (а также и более).
Наиболее часто он применяется по той простой причине, что большое количество команд ориентировано именно на операции с этим регистром (буква **W** входит в состав этих команд).
Например, для того чтобы перегрузить данные из одного регистра в другой, сначала данные из регистра-отправителя загружаются в регистр **W**, а затем из **W** загружаются в регистр-получатель, так как команды "прямой переправки" байта из одного регистра в другой, в списке команд нет.

Флаги

Определение флагов дано выше.
Основная часть флагов находится в регистре **STATUS**.
Самый часто применяемый из них - **флаг нулевого результата** (**2**-й бит с названием **Z**).
Он показывает, является ли результатом проведенной операции нулевой результат (флаг поднят, что соответствует установке **2**-го бита регистра **STATUS** в **1**) или этот результат не нулевой (флаг опущен: во **2**-м бите - **0**).
Кроме информирования программиста о результате операции, содержимое бита **Z** может являться критерием выбора того или иного сценария работы программы, для чего он чаще всего и применяется.
Например, программист поставил условие: если результат операции не равен 0, то программа исполняется далее, а если равен нулю, то происходит переход на выполнение подпрограммы (или на метку) **ABCD**.
После выполнения операции, флаг или останется опущенным (**0**) или поднимется (**1**).
Далее следует команда ветвления типа: если **2**-й бит регистра **STATUS** не равен **0**, то программа исполняется дальше, а если равен **0**, то происходит переход на выполнение

подпрограммы (или на метку) **ABCD**.
Дешево и сердито.
Если использовать цепочку таких проверок, то можно, например, выяснить, попадает ли результат операции в какой-то числовой "сектор" или нет, равен ли он какому-то числу или нет, и в зависимости от этого, направить программу в то или иное "русло".
Не трудно заметить, что анализ содержимого бита **Z** (а также и некоторых других флагов), происходит при выполнении так называемых **команд ветвления**, суть которых: **если X, то так, а если Y, то по другому**, применяющихся при выборе так называемых "сценариев работы программы".
Это словосочетание также, как и словосочетание "рабочая точка программы" - моя "самодеятельность", и я "ввожу его в эксплуатацию" с целью наиболее комфортного восприятия информации.
Сценарии работы программы это возможные варианты ее исполнения, которые "привязаны" к либо к выполнению, либо к невыполнению каких-то условий.
Необходимо также отметить то, что при использовании одной команды ветвления, возможных сценариев может быть только 2, а при применении нескольких - количество возможных сценариев будет кратным двум.
Флаги переноса - заёма **C** (**0**-й бит регистра **STATUS**) и десятичного переноса - заёма **DC** (**1**-й бит) показывают, было ли переполнение байта или младшего полубайта байта (извиняюсь за тавтологию) соответственно.
Переполнение это выход результата операции за пределы максимально возможного числа, определяемого разрядностью.
Например, наибольшее число в пределах байта (8 разрядов) → 255.
Если, например, складываются числа 200 и 100, то происходит переполнение, и флаг **C** устанавливается в 1 (поднимается).
Кроме того, при выполнении команд сдвига, флаг **C** загружается старшим или младшим битом байта сдвигаемого регистра (в зависимости от направления сдвига).
На флаги **переполнения сторожевого** таймера и **включения питания** пока можете не обращать внимания.
Они используются достаточно редко.
То же самое относится и к флагам **EEIF** и **WRERR** регистра **EECON1** (см. распечатку).
Флаги **TOIF**, **INTF**, **RBIF** регистра **INTCON** работают по единому принципу.
Отличия только в источниках прерываний.
Перед работой с ними, их необходимо сбросить программно.
"Разборки" с этим сбросом будут позднее.
С содержимым регистров специального назначения, в основном, разобрались.
Пошли дальше.

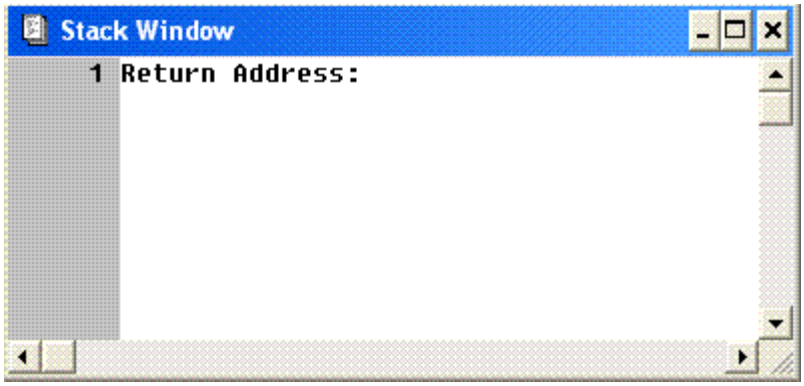
Метки

Само название уже отражает смысл этого понятия.
Если нужно осуществить переход на выполнение команды программы, отстоящей от команды начала перехода на какое-то количество команд и не являющейся первой командой подпрограммы, то эта команда помечается меткой (на "статус" подпрограммы "не тянет").
Название метки представляет собой результат фантазии программиста.
Метка ставится в специально отведенное для нее (них) "место" в тексте программы.
Она должна указывать на ту команду, на которую нужно осуществить переход.
Принципиальной разницы между метками и названиями подпрограмм нет и различия между ними достаточно условны.
Название подпрограммы указывает на первую команду подпрограммы, а метка помечает команду, отличную от первой команды подпрограммы.
Вот и все различие.
Техническая сторона переходов на метки и переходов в подпрограммы → одна и та же.
Самое главное - понимать смысл и "технологию" такого рода переходов, а проклассифицировать их можно на свое усмотрение.

Аппаратный стек

Является всё тем же оперативным запоминающим устройством "объемом" в 8 ячеек по 13 битов каждая.
Он поддерживает команды переходов/возвратов по условию (условные переходы), а также и переходы/возвраты, связанные с прерываниями.

Это означает то, что существует небольшая группа специальных команд, после исполнения которых происходит либо "закладка" в стек адреса команды, следующей после такой "специальной" команды, с так называемым "переходом по стеку" на начало исполнения некой программной процедуры, либо возврат на эту "следующую" команду (после исполнения этой "некой" процедуры), с использованием другой, "специальной" команды возврата (возврат по стеку).
В последнем случае, ранее "заложенный" в стек адрес команды из него "извлекается".
Он и является "указателем места", в которое должен быть осуществлен возврат.
Соответственно, эти специальные команды разделяются на команды перехода по стеку и возврата по стеку.
Для того чтобы осуществить переход по стеку на исполнение какой-то программной процедуры и возврат по стеку после ее (процедуры) исполнения, нужны 2 такие специальные команды: сначала (по ходу исполнения программы) команда перехода по стеку, а затем - команда возврата по стеку.
Если речь идет о безусловных переходах, то такого разделения нет, так как, при помощи одной и той же команды безусловного перехода, можно и "перейти", и "вернуться" в любое "место" программы (на любую ее команду) без задействования стека.
Вывод: главная функция стека - хранение адреса возврата в течение всего времени исполнения программной процедуры.
В упрощенном виде, он представляет собой некую таблицу из одного столбца и восьми строк.
Изначально (на момент начала программы), эта таблица пустая.
В MPLAB, это выглядит так:



Если, по ходу исполнения программы, был осуществлен условный переход, то в верхнюю строку (так называемая **вершина стека**), из счетчика команд PC, копируется 13-ти разрядное слово (8 бит **PCL** и 5 бит **PCH**) адреса следующей, после команды условного перехода, команды (для дальнейшего возврата программы по этому адресу).
Если до возврата из первого условного перехода, исполняется еще одна команда условного перехода, то адрес возврата первого условного перехода сдвигается на одну позицию (строку) вниз, занимая вторую строку, а адрес возврата более "позднего" условного перехода помещается в первую строку таблицы (в вершину стека).
Таким образом, стек можно заполнить 8-ю адресами возвратов, после чего наступает его переполнение, чего допускать нельзя (указателя переполнения стека нет), так как будет "Гитлер капут".
То есть, в стек может быть заложено не более 8-ми адресов возврата, и программист должен за этим следить при составлении программы.
Даже в достаточно сложных программах, работа стека на грани переполнения - редкое явление и его "емкости" хватает с избытком.
Это объясняется тем, что по мере работы программы, адреса возвратов из стека извлекаются, освобождая свободное место.
Адреса возвратов извлекаются только из вершины стека, а остальные 7 его строк используются как оперативная память для остальных адресов возврата, которые либо сдвигаются к вершине стека по ходу извлечения из него адресов возвратов, последовательно занимая вершину стека и "уходя" из него, либо наоборот, сдвигаются вниз от вершины, при выполнении новых команд условных переходов.
И то, и другое может чередоваться, что предполагает определенную последовательность как закладки адресов возвратов в стек, так и их извлечения их из него.
Спешу успокоить начинающих: на первых порах достаточно только одной вершины стека, то есть, первой (верхней) его строки.
В нее можно последовательно заложить и извлечь хоть сотню адресов возвратов.
Естественно, в этом случае, нужно позаботиться о том, чтобы следующий условный переход осуществлялся только после возврата из предыдущего условного перехода.

Если Вы это освоите, то со временем, Вы естественным образом перейдете к более сложным манипуляциям с условными переходами.
Это произойдет тогда, когда Вы захотите сделать свою программу более компактной (с меньшим количеством команд).
Некоторые программы вообще не содержат условных переходов (стек не задействован), а в некоторых случаях без них никак не обойтись.
Подробнее о работе со стеком - в следующих разделах.
Итак: ***микроконтроллер это набор многофункциональных устройств.***
Программа определяет конкретные функции этих устройств и их режимы работы на различных участках выполнения программы (см. регистры специального назначения).
На регистрах общего назначения строятся различные устройства, применяемые в цифровой технике. Таковыми устройствами их делает программа.
И в том, и в другом случае именно программа (программист) определяет все то, что будет происходить с этими элементами ПИК-конструктора, и поэтому далее речь пойдет о "начинке" программы, то есть о командах.
Примечание: то, что Вы прочитали, есть основа, с еще не достаточно четко "прорисованными" деталями.
Одноразово выдавать Вам весь "массив", связанной с этим "ПИК-конструктором", информации, было бы абсолютно неправильным и опрометчивым решением.
"Въезжать" в этот "массив" нужно постепенно, с соблюдением "техники безопасности" и не делая "резких движений".
Это и есть дальнейшая стратегия "въезда".

3. Система команд PIC16F84A

Вашему вниманию предлагается описание команд ассемблера для PIC контроллеров.

Таблица команд ассемблера для ПИКов находится в *"Приложении №10"*.

Команды разделены на 3 группы:
- ***байт-ориентированные команды***, то есть команды, производящие действия с байтами,
- ***бит-ориентированные команды***, то есть команды, производящие действия с битами,
- ***команды управления***, то есть команды, осуществляющие переходы и возвраты, а также команды сброса сторожевого таймера WDT и перехода в спящий режим SLEEP.

Из списка команд управления можно смело вычеркнуть последние 2 команды, так как они могут быть использованы только для поддержки совместимости PIC16F84A с более "древним" семейством микроконтроллеров PIC16C5x, а для совместимости PIC16F84A с последующими, выпускаемыми микроконтроллерами, эти команды разработчики вообще не рекомендуют использовать.
Итого: осталось 35 команд.
Разве это много?
По сравнению, например, с изучением иностранного языка, это вообще мизер.
Давайте с ними разбираться.
Сначала - информация по таблице команд.
В первой колонке располагается название команды, а в рамке под этим названием - краткое содержание команды.
Сразу обращаю Ваше внимание на то, что подчеркивание команды есть только элемент ее выделения, и ничего более. В тексте программ команды не подчеркиваются.
Латинские буквы в конце команды обозначают:

f - название регистра (в общем виде), к которому эта команда обращается, и с содержимым которого будет производиться действие. Иными словами, наличие в команде буквы **f** означает, что команда обращается к содержимому регистра специального или общего назначения, с которым программист хочет произвести определяемое этой командой действие.
d - указатель места сохранения результата при операциях с байтами. Если d = 0, то результат исполнения команды сохраняется в регистре **W** (в аккумуляторе). Если d = 1, то результат исполнения команды сохраняется в регистре **f**, то есть, в том же самом регистре, с содержимым которого производится действие.
b - номер бита в регистре, с которым производится действие (для бит-ориентированных команд).
k - константа, то есть некое число в пределах байта, которое задается программистом.

Во второй колонке, находится более подробное описание действия, которое производит команда.
В третьей колонке, даны примеры выполнения команд.
В четвертой колонке, указано количество машинных циклов, в течение которых данная команда исполняется.
Обратите внимание на то, что большая часть команд исполняется за 1 машинный цикл. Имеются также команды, выполняющиеся за 2 машинных цикла (в основном это команды управления).
Комбинация числа 1 и двойки в скобках, означает, что по одному сценарию выполнения команды, она выполняется за один машинный цикл, а по другому - за два.
Напоминаю, что подобного рода команды (команды ветвления) имеют только 2 сценария исполнения команды.
В пятой колонке указаны флаги, которые "реагируют" на выполнение данной команды. Ячейки пятой колонки команд, на выполнение которых флаги не реагируют, закрашены серым цветом.
Сразу обратите внимание на следующую особенность, знание которой поможет Вам при запоминании команд: значительная часть команд состоит из 5-ти букв, причем корень слова

команды, характеризующий ее суть, состоит, в большинстве случаев, из 3-х букв, а последние две буквы показывают, откуда и куда копируется (переписывается) число или с какими регистрами нужно произвести определяемое командой действие.
Например, команду **ADDWF** можно условно разделить на 2 части: корень **ADD**, который обозначает операцию сложения, и **WF**, где первая буква обозначает регистр **W** (аккумулятор), а вторая - регистр **F**, то есть регистр, к которому обращается команда.
Следовательно, из самого названия команды становится понятным, что складывается содержимое регистров **W** и **F**.
В качестве этого условного регистра **F**, используются регистры общего или специального назначения.
Например, если в "шапке" программы определено название какого-нибудь регистра общего назначения как **ABCD**, и по ходу исполнения программы, в него было записано какое-нибудь число, которое необходимо сложить с числом, находящимся в аккумуляторе (**W**), с сохранением результата сложения в том же регистре **ABCD**, то это будет выглядеть так:

```
addwf       ABCD,1
```

При этом следует отметить, что в регистр **W** программист может переписать содержимое любого другого регистра, и по своей сути, такого рода операция есть операция сложения чисел, находящихся в двух регистрах (через регистр **W**), с выбором места сохранения ее результата.
Рассмотрим команду **ADDLW**:
ADD - операция сложения, **L** - признак того, что в ней участвует константа, **W** - признак того, что в ней участвует регистр - аккумулятор.
Например:

```
addlw       k
```

Это означает то, что складывается содержимое регистра **W** и некая константа, заданная программистом.
Если она равна, например, числу 3, то это будет выглядеть так:

```
addlw       3
```

Так как константа не является регистром, то место сохранения результата операции однозначно - регистр **W**.
Другой пример: операция копирования содержимого одного регистра в другой.

```
movwf       ABCD
```

Расшифровка: **MOV** - обозначение команды копирования.
Копирование данных производится из регистра **W** в регистр **F** (то есть, в регистр **ABCD**).
Так как никакой логической операции при этом не совершается, то и вопроса о сохранении результата неосуществленной логической операции не возникает.
Просто происходит банальное копирование данных в указанный в команде регистр.
Обращаю Ваше внимание на то, что в системе команд для ПИКов, при простом перемещении данных из одного регистра в другой (то есть при операции без логических действий), речь идет именно о копировании, без вырезания данных из регистра - отправителя.
Конечным результатом этой операции является числовое равенство содержимого обеих регистров.
Аналогично:
AND - операция побитного "И" содержимого двух регистров,
CLR - сброс в ноль содержимого регистра,
COM - инверсия всех битов регистра,
DEC - декремент содержимого регистра (минус 1),
INC - инкремент содержимого регистра (плюс 1),
IOR - побитное "ИЛИ" содержимого двух регистров,
NOP – "пустой" машинный цикл (ничего не происходит),
RLF - циклический сдвиг содержимого регистра влево,
RRF - циклический сдвиг содержимого регистра вправо,
SUB - операция вычитания,
XOR - побитное "исключающее ИЛИ" содержимого двух регистров,
BTF - команды ветвления по значению выбранного в регистре бита,
RET - возврат с загрузкой константы в W,
4 буквы в корне команды:
SWAP - поменять местами полубайты регистра,
CALL - переход по условию (с задействованием стека),

GOTO - переход без условия,
5 букв в корне команды:
SLEEP - уход в режим пониженного энергопотребления,
6 букв в корне команды:
RETURN - возврат из подпрограммы (кроме подпрограмм обработки прерываний),
RETFIE - возврат из подпрограммы обработки прерываний.
1 буква в корне команды:
B - операции с битом. Имеются ввиду команды **BCF** и **BSF**.
Вторая буква в этих командах является признаком операций с битом:
- **C** - установить бит в **0**,
- **S** - установить бит в **1**,
- **F** - признак регистра.

Получается: сбросить (**0**) или установить (**1**) какой-то из битов регистра, к которому обращается команда.
С учетом сказанного выше, и того, что таблица команд содержит достаточно подробную информацию, я остановлюсь на наиболее трудно понимаемых командах.
На данном этапе обучения, от Вас требуется только общее понимание смысла действий, производимых командой и заучивание их названий.
В дебри лезть незачем, иначе навредите сами себе.
"Хозяином" этих команд Вы станете позднее - тогда, когда пойдет работа по составлению программ, а пока "вживайтесь" в команды ассемблера.
На первых порах, наибольшие трудности, обычно, вызывают команды ветвления.
Сразу следует уяснить, что ветвления, то есть выбор какого-то одного из двух сценариев работы программы, могут быть результатом выполнения различных действий.
Байт – ориентированные команды ветвления: DECFSZ, INCFSZ.
В подавляющем своем большинстве, эти команды обращаются к регистрам общего назначения.
Самый распространенный случай: внутри полного цикла программы, имеется еще один внутренний цикл, и рабочая точка программы должна "прокрутиться" по этому внутреннему циклу, например, 10 раз, чтобы после этого выйти на полный цикл программы.
В качестве счетчика такого рода "проходов" ("колец"), назначается какой-нибудь из регистров общего назначения.
Назовем его, например, **XYZ**.
В начале полного цикла программы, в него записывается число "проходов", то есть, число **10**.
Предположим, что программа работает и ее рабочая точка вышла на 1-й внутренний цикл.
А в его конце стоит команда **DECFSZ**, в результате действия которой, содержимое регистра **XYZ** уменьшается на единицу и становится равным **9**-ти.
Эта девятка сохраняется в том же регистре **XYZ**.
Так как это число не равно 0, то происходит переход на начало 2-го внутреннего цикла программы и все повторяется снова до тех пор, пока, после очередного вычитания единицы, в регистре **XYZ** не установится число **0**.
Как только это произойдет, будет осуществлен переход не на начало следующего внутреннего цикла программы, как это было ранее (первый сценарий работы программы), а на второй сценарий работы программы, который, в общем виде, называется **"программа исполняется далее"**.
По этому сценарию, в пределах дозволенного, могут осуществляться какие угодно действия (определяется замыслом программы).
Из этого примера видно, что происходит то, что называется ветвлением программы.
Еще один пример.
К выводам порта B подключен 7-сегментный индикатор.
Полный цикл программы равен, например, 1 сек., а его сегменты высвечиваются в течение 0,1 сек., а остальное время погашены. Требуется увеличить это соотношение.
Следовательно, необходимо "встроить" ("врезать") в тот участок программы, в котором индикатор активен, подпрограмму задержки, функция которой заключается в "закольцовке" рабочей точки программы внутри этой подпрограммы.
А как же иначе, ведь рабочая точка программы всегда должна находиться в движении?
Вот и пусть она "мотает" заданное программистом количество "кругов" в подпрограмме задержки, ведь время-то идет...
Такие подпрограммы имеют в своей основе все те же команды **DECFSZ** (**INCFSZ**), только

безусловные переходы при этом гораздо "скромнее" - всего на несколько команд, а не на десятки, то есть, рабочая точка программы, на время задержки, "крутится" по относительно малому (с малым количеством команд) кольцу подпрограммы задержки.
Выглядит это так.
Подпрограмма задержки, в простейшем случае, начинается с команды **DECFSZ** (**INCFSZ**) и обращается к регистру общего назначения, в который, перед ее началом (в промежутке времени между моментом активации индикатора и до начала подпрограммы задержки), закладывается некое число (константа), которое и будет определять время задержки (количество "колец").
Предположим, что в регистр **XYZ** заложено число (константа) 250.
СтОит только рабочей точке программы войти в подпрограмму задержки, она не выйдет из нее до тех пор, пока задержка не будет полностью отработана.
Смысл этой "закольцовки" заключается в том, что при выполнении, например, команды **DECFSZ** и отсутствии нулевого результата операции декремента, переход, в конечном итоге, происходит на все ту же команду **DECFSZ** и все повторяется по новой, пока число 250 не уменьшится до нуля, а это требует времени.
Вот Вам и задержка.
При нулевом результате операции, рабочая точка программы выходит из подпрограммы задержки, после чего индикатор переводится в пассивное состояние и программа исполняется далее.
Зная длительность машинного цикла, можно точно просчитать время задержки.
Этот пример, конечно, простейший, так как для относительно больших задержек, при длительности машинного цикла = 1мкс (кварц 4 Мгц.), одного регистра не достаточно (необходимо несколько регистров с последовательным их декрементом).
С этим мы разберемся позднее.
Сейчас главное - прочувствовать "кольцевую природу" программ и представить себе, как в главном "кольце" (цикле) программы, образуются большие и маленькие "кольца", и как рабочая точка программы в них "входит, выходит, идет по прямой, заходит в другое кольцо" и т.д.
Примечание: все, сказанное выше, относится и к команде **INCFSZ**.
Отличие - только в направлении счета (+1).
Оставшиеся две команды ветвления **BTFSC** и **BTFSS** ориентированы на операции с битами регистров.
При применении этих команд, никаких действий с байтами не производится.
Критерием ветвления является значение выбранного бита (**0** или **1**).
Например, если на момент исполнения команды **BTFSC**, значение выбранного бита регистра **XYZ** (например, **бита №2**) равно **1**, то в соответствии с логикой команды **BTFSC**, выполняется следующая команда, а если значение этого бита равно **0**, то вместо следующей команды, исполняется "пустышка", в виде команды **NOP** (нет действия или операции. Задержка на 1 м.ц.).
После этого "пустого" машинного цикла, выполняется следующая команда (вторая после **BTFSC**).
Обращаю Ваше внимание на то, что в этом случае, команда **NOP** как бы "виртуальна" (формируется аппаратными средствами ПИКа), так как в тексте программы команды **NOP** нет, хотя, по факту, она и заменяет, в случае равенства **2**-го бита нулю, следующую, после **BTFSC** , команду.
Получается, что в этом случае, команда присутствует в тексте программы, но не исполняется (вместо нее, исполняется **"виртуальный" NOP**).
Итоговый смысл:

- **в случае равенства 2-го** (в данном случае) **бита единице, исполняется следующая, после команды BTFSC, команда,**
- **в случае равенства 2-го** (в данном случае) **бита нулю, происходит как бы "скачёк" через эту "следующую" команду (посредством исполнения, вместо нее, "виртуального" NOPа).**

Естественно, что никакого "скачка" через команду не происходит, ведь **"виртуальный" NOP** исполняется, но впечатление "скачка" создается.
Кстати, это очень полезное впечатление, от которого не нужно отмахиваться, а нужно использовать в своей работе (очень удобно), не забывая при этом про **"виртуальный NOP.**
Примечание: словосочетания **"виртуальный NOP"** Вы нигде не найдете, так как это моя

фантазия. Слово "виртуальный" означает, что то, на что указывает это слово, исполняется не программно, а аппаратно.
Ведь нужно же, "для порядка", кратко и емко "обозначить это явление"?
В соответствии со сказанным, время исполнения команды **BTFSC**, в зависимости от сценария ее исполнения, будет составлять **либо 1, либо 2 машинных цикла**.
Вот Вам и объяснение того, что Вы видите в 4-м столбце таблицы команд, в строке с командой **BTFSC** {**1(2)**}.
То же самое, с учетом специфики команд, относится и к остальным командам ветвления (**DECFSZ, INCFSZ, BTFSS**).
А если, после команды **BTFSC**, исполняется команда безусловного перехода, как это и происходит во многих случаях?

Пример самого распространенного варианта:

```
btfsc      XYZ,2
goto       Cycle
-----      -----
```

1-й сценарий: если **2-й бит** регистра **XYZ** равен **1**, то происходит безусловный переход в подпрограмму, например, **Cycle**.
2-й сценарий: если **2-й бит** регистра **XYZ** равен **0**, то перехода в подпрограмму **Cycle** не происходит (вместо этого - **"виртуальный NOP"**) и программа исполняется далее (дальнейшие команды программы обозначены `---- ----`).

Еще один вариант:

```
btfsc      XYZ,2
goto       Cycle
goto       Mem
```

1-й сценарий: если **2-й** бит регистра **XYZ** равен **1**, то происходит безусловный переход в подпрограмму **Cycle**.
2-й сценарий: если **2-й** бит регистра **XYZ** равен **0**, то происходит безусловный переход в подпрограмму **Mem**. Команда **goto Cycle** игнорируется (вместо нее исполняется **"виртуальный NOP"**).

Еще один вариант:

```
btfsc      XYZ,2
movwf      PortB
-----      -----
```

1-й сценарий: если **2-й бит** регистра **XYZ** равен **1**, то содержимое аккумулятора (**W**) копируется в **защелки порта B**.
После этого копирования, программа исполняется далее.
2-й сценарий: если **2-й бит** регистра **XYZ** равен **0**, то копирования содержимого аккумулятора в **защелки порта B** не происходит (вместо этого - **"виртуальный NOP"**), и программа исполняется далее.
В первых двух вариантах, используются переходы, а в последнем, не используются.
В любом из этих случаев, происходит ветвление, так как имеется **два сценария** работы.
Все, что сказано выше, относится и к команде **BTFSS**.
Единственное различие в том, что критерий анализа меняется на противоположный (сценарии меняются местами).
Начинающим, в первую очередь, необходимо обратить внимание на команду безусловного перехода **GOTO**.
Сначала нужно освоить ее, а потом переходить к изучению команды условного перехода **CALL**.
В принципе, при составлении программ, можно вообще не применять команду **CALL**, а пользоваться только командой **GOTO** как по своему прямому назначению, так и заменяя ей команду **CALL**.
Обе эти команды, "стратегически", осуществляют переход по одному и тому же принципу (типа "куда захочу, туда и перейду"), но есть "тактические" отличия.
Команда **GOTO** "гуляет сама по себе" (стек не задействован), а команда **CALL** - только в "связке" с командой возврата **RETURN** или **RETLW**.

А как же иначе, ведь после отработки той подпрограммы, в которую осуществлен условный переход, для обеспечения возврата "в то место, откуда пришел", из стека, нужно "выгрузить" адрес возврата.
Принцип замены команды **CALL** на **GOTO** состоит в том, что, кроме команды **GOTO**, по которой происходит переход, необходима еще одна команда **GOTO**, которая организует возврат.
В случае организации условного перехода (**CALL**), возврат происходит на команду, которая не помечена меткой.
Следовательно, при замене условного перехода на безусловный, необходимо выставить на команде, на которую нужно вернуться, метку, а в рабочей части команды возврата (в данном случае, команды **GOTO**) указать эту метку.
В общем виде, это выглядит так:

```
                -----      -----
                goto       Cycle
Metka           xxxx       xxxx
                -----      -----
                -----      -----
Cycle           yyyy       yyyy
                -----      -----
                goto       Metka
                -----      -----
```

`----- -----` обозначены какие-нибудь команды программы.
Их может быть разное количество (зависит от программы).
По команде безусловного перехода **GOTO**, сначала осуществляется переход в подпрограмму **Cycle**.
Следующая, после **GOTO,** команда, обозначенная **xxxx**, для обеспечения дальнейшего возврата, помечается меткой **Metka**.
Происходит переход в подпрограмму **Cycle**, и она, начиная с первой ее команды, обозначенной **yyyy**, исполняется до своей последней команды, в качестве которой используется все та же команда **GOTO** (goto Metka), но на этот раз, она обеспечивает возврат.
После этого, происходит возврат на команду, выделенную меткой **Metka**, и программа исполняется далее.
Между **Metka** и **Cycle** может быть, например, 10, 50, 100 или другое количество команд, в которых также могут быть организованы переходы и возвраты.
Таким образом можно организовать переходы и возвраты без использования команды **CALL**, то есть без работы со стеком.
Результат - один и тот же.
Рекомендую начинающим, на первых порах, поступать именно так, а со стеком поработаете тогда, когда переходы и возвраты перестанут вызывать у Вас затруднения.
Для любознательных: описанная выше процедура, для случая условного перехода, выглядит так:
Первую сверху команду **GOTO** нужно заменить на команду **CALL**.
Метка **Metka** не нужна.
Нижняя команда **GOTO** заменяется на специальную (для работы со стеком) команду возврата **RETURN** (точку возврата указывать не нужно, так как ее адрес "лежит" в вершине стека).
Команда **RETLW** также работает со стеком.
От команды **RETURN** она отличается только тем, что, при возврате, осуществляется действие: в регистр **W** загружается указанная константа.
Команда **RETURN** никаких действий, кроме возврата по стеку, не производит.
Команда **RETFIE** применяется только для возврата из подпрограммы прерываний, о которой будет отдельный разговор.
Команда **CLRWDT** используется для периодического, программного сброса сторожевого таймера **WDT**, если он включен.
В отличие от команды **MOVWF**, в которой все ясно и понятно (копирование байта из аккумулятора в указанный регистр), команда **MOVF** может вызвать у начинающих некоторые затруднения.

Команда **MOVF** копирует содержимое указанного регистра либо в регистр **W** (аккумулятор), либо в тот же, указанный регистр.
В основном, сохранение происходит в регистре **W** (команда **MOVF** копирует, в регистр **W**, содержимое указанного в команде регистра.
Казалось бы, "бестолковое" копирование содержимого, указанного в команде регистра, в этот же регистр, все-таки имеет практический смысл: можно проверить, равен или нет нулю результат этой операции, а затем "разветвиться" на 2 сценария.
Если содержимое регистра равно 0, то флаг нулевого результата **Z** поднимется (установится в **1**), а если нет - опустится (установится в **0**).
"Манипуляции" с флагом **Z** будут рассмотрены позднее.
Чаще всего команда **MOVF** применяется для считывания данных, с выводов портов (регистры **PORTA** и **PORTB** это регистры специального назначения), в регистр **W**.
Выглядит это так: **movf PortA,W** или **movf PortB,W**.
В дальнейшем, с целью дальнейшей обработки результата этого считывания, можно скопировать содержимое регистра **W** в один из регистров общего назначения.
Например, для выяснения состояния клавиатуры, подключенной к выводам порта.
Еще раз обращаю Ваше внимание на то, что напрямую скопировать данные, из одного регистра в другой, нельзя.
Их (данные) нужно сначала скопировать, из регистра – адресата, в буферную память (в регистр **W**), а затем скопировать данные из нее, в регистр - получатель.
Для случая копирования данных из регистра **PortB** в регистр общего назначения **XYZ**, это выглядит так:

```
movf    PortB,W
movwf   XYZ
```

Буквально: скопировать содержимое регистра **PortB** в аккумулятор, а затем скопировать содержимое аккумулятора в регистр **XYZ**.
Или проще: переслать содержимое регистра **PortB** в регистр **XYZ**.
Примечание: в приведенных выше и ниже примерах использования команд, я не придерживаюсь правил, используемых при написании программ.
Синтаксические правила написания программ будут изложены позднее, и этим правилам Вы будете обучаться в специализированном редакторе программы **MPLAB**.
Стандартные логические операции с корнями **AND** ("И"), **IOR** ("ИЛИ"), **XOR** ("исключающее ИЛИ"), **COM** (инверсия) в особых пояснениях не нуждаются (это из основ цифровой техники).
При выполнении команд циклического сдвига **RLF** или **RRF**, происходит циклический сдвиг влево или вправо содержимого регистра, к которому обращается команда сдвига.
Циклический сдвиг происходит **через флаг переноса - заёма** (нулевой бит регистра **STATUS**), с названием **C**.
Эти команды используются, например, при преобразованиях кодов, при организации сложных арифметических действий и т.д.
При организации этих процедур, используется также и команда **SWAPF**.
Начинающим советую пока, на время, "отложить последние 3 команды в сторону" и особо на них не "зацикливаться".
"Разборки" с ними будут позднее.
Команда **NOP** (нет операции) на первый взгляд может показаться не слишком значимой, но это не так.
При составлении программ, достаточно часто встречаются ситуации, когда необходимо осуществить небольшую задержку выполнения какой-то последующей команды, не производя при этом никаких операций с числами, или ситуации, когда необходимо сделать время исполнения обеих сценариев команды ветвления одинаковыми, не производя при этом никаких операций, и т.д.
В этих случаях, используется команда **NOP**.
Если **NOP**ы вставляются в "линейный" участок программы, то каждый **NOP** это задержка на **1 машинный цикл**, а если **NOP**ы "вставляются" в циклические подпрограммы, то в зависимости от их "конструкции", время исполнения программы, в итоге, может увеличиться на десятки, сотни и даже тысячи машинных циклов.
Примечание: я постарался объяснить Вам только то, что необходимо для "старта".
Детальные "разборки" с командами последуют далее, когда начнется "живая" работа с текстами программ и когда будут "подворачиваться" удобные поводы.
На мой взгляд, теоретическое обучение есть обучение, хотя и совершенно необходимое, но

какое-то "недоношенное", и которое, по своей эффективности, "в подметки не годится" обучению на "живых" примерах.

Дополнительно.

Оговорка: если Вы только начинаете и не имеете опыта, то изложенная ниже информация, вернее всего, будет восприниматься "туговато", но, все-таки, я советую Вам бегло с ней ознакомиться с целью того, что называется "быть в курсе".
Потом, когда поднаберетесь опыта, можно разобраться с ней более детально.

Об ошибках, допущенных в "фирменном" описании системы команд среднего семейства м/контроллеров PICmicro.

("Справочник по среднему семейству м/контроллеров PICmicro", перевод технической документации DS33023A компании Microchip Technology Incorporated, ООО "Микро-Чип",2002)

Пояснения:

Лично я, "въезжал" в команды при помощи "разборок" с проверенными в работе текстами программ и пользовался "фирменной" распечаткой команд только в части касающейся самих команд и краткого их названия, без детальных разбирательств с примерами применения команд, приведенными в этой распечатке: все примеры я брал из "живых" текстов программ.
Когда возникла необходимость загрузки на сайт файла таблицы команд, я, доверяя профессионалам, без всякой "задней мысли", просто скопировал примеры применения команд из "фирменной" технической документации, особо не задумываясь о том, что что-то в ней может быть "криво".
Основанием для такой легкомысленности являлось то, что у меня просто не было практической необходимости в "разборках" с примерами применения команд этой таблицы по той простой причине, что с этими примерами я детально разбирался, как говорится, "в живую", работая непосредственно с текстами проверенных в работе программ, что гарантирует максимальное качество этих детальных "разборок".
Так это "святое неведение" и продолжалось бы, если бы не вопросы **Дмитрия Дубровенко**.
Он попытался детально разобраться с примерами "фирменной" таблицы и "заработал" себе "головную боль" в виде вопросов, с которыми он ко мне и обратился.
Вот уж действительно внимательный и дотошный человек. Побольше бы таких.
Я наконец-то удосужился "въехать" в эти примеры, и у меня "волосы встали дыбом": никак не ожидал от профессионалов такого подвоха.
Вот уж воистину "доверяй, но проверяй".
Ниже, я постараюсь объяснить, в чем тут дело и привести таблицу команд в "божеский вид".
Исходные данные:

"Кривая", "фирменная" таблица команд (та, что раньше "лежала" в этом разделе), находится в *"Приложении №10"*.

Вопросы **Дмитрия Дубровенко** выделены **синим цветом**.

1. По команде **ADDWF**, использующейся для вычисляемых переходов: ***"В примере указано ADDWF PCL,0 , то есть, результат должен записываться в аккумулятор, а не в PCL. Если это какая-то особенность, у Вас нигде не сказано про это (что надо ставить 0, а не 1)".***
Ну конечно же, нужно писать **PCL,1** , а не **PCL,0**, а иначе приращения счетчика команд **PC** (вычисляемого перехода) не произойдет.
Ошибка в "чистом виде", причем, повторяющаяся в двух примерах для команды **ADDWF**.
Почему в моих дальнейших объяснениях (в последующих разделах) **ADDWF PC,1** , а не **ADDWF PCL,1**?
А это зависит от того, под каким именем "прописан" регистр счетчика команд в "шапке" программы: можно "прописать" и так: **PC**, тогда **ADDWF PC,1**, а можно и по-другому: **PCL**, тогда **ADDWF PCL,1**).
"Это на любителя".
В примерах таблицы для команды **ADDWF**, название **PCL** указано потому, что в комментариях (в последнем примере) указан **PCH** (для обозначения "составных частей" **PC** и порядка их "взаимодействия", а точнее, если речь идет о вычисляемом переходе, "невзаимодействия", так как при переполнении **PCL**, приращения **PCH** не происходит).

2. По командам **RLF** и **RRF** (примеры 2 и 3): ***"Написано RLF INDF,1 (RRF INDF,1). Значение W, до выполнения - неопределенное. Почему, после выполнения команды, оно становится 17h?"?***
Ответ: регистр **W** "тут вообще не при чём", так как при циклическом сдвиге (в любую сторону), он не задействуется и, по этой причине, операция циклического сдвига, по определению, никак не может повлиять на содержимое регистра **W**: что в нем "лежало до того", то и будет "лежать после того".
Обращение команды к содержимому регистра **INDF**, если результат сохраняется в нем же (**INDF,1**), также не влияет на содержимое регистра **W**.
Регистр **W** нужно вообще убрать из комментариев 2-го и 3-го примеров для команд **RLF** и **RRF**, что я и сделал.
Как **xxxxxxxx** "превратилось" в **17h**?
"Тайна сия велика есть". Если кто-то знает, то поделитесь разгадкой.

3. По команде **ADDLW** и другим командам, где упоминается **HIGH(LU_TAB)**:
"До выполнения: W=0x10, LU_TAB=0x9375 (адрес в памяти программ). После выполнения: W=0xA3. Откуда берется этот адрес (он ведь больше максимального количества команд), и почему операции производятся со старшим байтом?"
Несоответствие (ошибка) обнаружено совершенно верно: для наибольшего "объема" памяти программ ПИКов, величиной в 8 килослов, максимальная величина адреса памяти программ составляет **1FFFh**.
9375h больше 1FFFh, и это число (9375h) использовать нельзя (в памяти программ ПИКов нет таких адресов).
Если предположить, что речь идет не о адресе, а о содержимом 2-хбайтного регистра (9375h = 10010011 01110101), то, опять же, имеет место быть ошибка по той простой причине, что комментарий "адрес в памяти программ" не верен.
По всей видимости, составители подобного рода примеров пытались показать работу с табличными данными, но сделали они это "неуклюже", в примитивном виде, да еще и с ошибками и "недомолвками".
Ничего вредоносней таких "недоношенных" примеров и представить себе трудно, и поэтому я заменил их на примеры работы с заранее "прописанной", в "шапке" программы, константой **CONST** (этому названию или другому названию, директивой **EQU**, можно присвоить любое числовое значение в диапазоне **0 … 255**).

Пояснения

По командам управления:
"11 бит адреса загружаются из кода команды в счетчик команд PC(10:0). 2 старших бита загружаются в счетчик команд PC(12:11) из регистра PCLATH. Вот это совсем не понял."
В "Справочнике по среднему семейству м/контроллеров PICmicro" (можно скачать на сайте Микрочипа), на странице **6-5**, найдите рисунки **6-2б, в**.
При переходах, это именно та специфика адресации, которая имеет место быть, и ее просто нужно принять как данность.
По большому счету, на эти фразы можно просто не обращать внимания, так как влияния на эту специфику программист оказать не может.
Это аппаратная "епархия" ПИКа.
"Он там все сам разрулит" (при исполнении команд переходов/возвратов).
"Возможно ли использовать аккумулятор в качестве аргумента именно в командах типа ADDWF (т.е. ADDWF W,d)? Насколько я Вас понял, нельзя?"
Никакого практического смысла в этом действии нет (разве только задержка, но для этого есть **NOP**ы): обращение командой **ADDWF**, к содержимому **W**, не вызовет никаких изменений его содержимого (при любом значении **d**): что в нем "лежало до того", то и будет "лежать после того" (можете проверить в симуляторе).
"Какие еще регистры, кроме W, не надо прописывать в шапке?"
Когда речь идет о "прописке", то под этим подразумевается предварительное указание, в "шапке" программы, адресов используемых в программе регистров области оперативной памяти.
"Прописка" адресов этих регистров обязательна (то ли "напрямую", то ли при помощи директивы **INCLUDE**), иначе **MPLAB** "откажется" создавать HEX-файл.

Регистру же **W**, адрес ставится в соответствие автоматически (он имеет строго фиксированный адрес).
Другое дело, в каком виде к нему обращаться в "рабочей" части программы: то ли нулем (**d=0**), то ли при помощи его общепринятого названия (**W**).
Если нулем, то его можно не "прописывать".
Если **W**, то лучше, на всякий случай, "прописать".
Это было обязательным в самых первых версиях **MPLAB**, а в более поздних версиях (включая и рекомендованную мной), букве **W** автоматически ставится в соответствие **0.**
В этом случае, аккумулятор можно не "прописывать".
Это же относится и к символьному обозначению регистра (**F** / **d=1**).
Можете провести эксперимент: откройте текст программы **Multi.asm** и проассемблируйте его с "пропиской" **F** и без нее (в "рабочей" части программы укажите **F**, а не **0**).
В обеих случаях ассемблирование пройдет успешно.
Выводы делайте сами.
С целью наиболее простого и удобного "въезда" в команды, "фирменную" таблицу команд я довольно-таки основательно видоизменил (см. ***"Приложение №11"***).
Эти изменения можно отследить, если сравнить
"Приложение №10" с ***"Приложением №11"***.
Если есть неясности, то пишите (просьба четко сформулировать вопрос).
В конце концов, ведь должна же существовать максимально доходчивая и понятная таблица команд, а не "кроссворд, от которого крыша едет".

"Чудеса" продолжаются.

Валерий Галкин обнаружил еще три ошибки в "расшифровке" системы команд, опубликованной в "Справочнике по среднему семейству ...".
Все они связаны с неверными числовыми результатами выполнения команд.
Вопросы Валерия Галкина выделены синим цветом.

1. Команда ANDWF f,d

Правильно ли указано в третьем примере, что результат исполнения команды - число 0x15 (по моему, 0x12)?

Валерий прав: при побитном "И" числа **00010111** (17h) и **01011010** (5Ah), получается число **00010010** (12h), а не **00010101** (15h).

2. Команда IORLW k

Правильно ли указано во втором примере, что после выполнения команды, W = 0x9F (по моему, W = 0xBF)?

Валерий прав: при побитном "ИЛИ" числа **10011010** (9Ah) и **00110111** (37h), получается число **10111111** (BFh), а не **10011111** (9Fh).

3. Команда XORLW k

Правильно ли указано во втором примере, что после выполнения команды, W = 0x18 (по моему, W = 0x98)?

Валерий опять прав: при побитном "Исключающее ИЛИ" числа **10101111** (AFh) и **00110111** (37h), получается число **10011000** (98h), а не **00011000** (18h).
Выводы делайте сами.
Валерий, спасибо за обнаружение "мин на этом минном поле".

Таблица команд ассемблера, которая не содержит ошибок, находится в *"Приложении №12".*

4. Что такое программа и правила ее составления. Пример создания программы автоколебательного мультивибратора. Директивы.

Программа это та или иная последовательность команд, которая реализует замысленные программистом действия с числами.

Производя те или иные действия с заранее заданными или формируемыми, в ходе исполнения программы, числами и/или с данными, поступающими от внешних устройств (что, по сути своей, есть все те же числа), можно сформировать требуемые алгоритмы и сигналы управления внешними исполнительными устройствами, что и есть "кульминация" работы программы (то, ради чего и "затеивается вся эта свистопляска" с нулями и единицами).
Специфики работ по созданию устройств на дискретных элементах цифровой техники (К555 и т.д.) и работ по созданию устройств на микроконтроллерах, значительно отличаются друг от друга.
При создании устройств на дискретных элементах цифровой техники, конструктор, преимущественно, имеет дело с машинными кодами.
Если речь идет об устройствах, в состав которых входит большое количество корпусов м/схем, то такое конструирование весьма хлопотно и трудоемко.
Устройства получаются энергоемкими, громоздкими и габаритными.
На их разработку и изготовление тратится много сил, времени и денег.
В процессе конструирования устройств на м/контроллерах, конструктор, преимущественно, имеет дело не с машинными кодами, а с языком программирования ассемблер (или с каким-либо языком высокого уровня, но в конечном итоге, все сводится к ассемблеру), с помощью которого, задачи решаются гораздо проще и эффективнее.
Устройства, по схемотехнике, получаются очень простыми, компактными, экономичными и относительно дешевыми.
Задача желающего научиться программированию ПИКов, на первых порах, сводится к изучению его "начинки", изучению команд ассемблера и принципов построения нескольких фундаментальных подпрограмм.
Потом, когда Вы "войдете во вкус", все станет гораздо понятнее и интереснее (по себе знаю).
А пока, наберитесь терпения (а куда деваться? Выбора-то нет... В этой "епархии ничего с неба не падает).
Итак, ранее выяснилось, что любая программа должна строиться по "циклическому принципу", и ее рабочая точка может двигаться по командам программы либо последовательно, если нет команд переходов, либо совершить "скачёк" в то место программы, которое указывается в команде перехода (с возвратом и без возврата), либо "закольцеваться", на какое-то время, в той или иной подпрограмме, с возможностью последующего выхода из этой "закольцовки".
По ходу создания программы, программист может реализовать в ней различное количество "колец" (больших, средних, малых или еще каких-то).
Самым большим "кольцом" является **полный цикл программы**, который, при работе программы, исполняется снова и снова, до тех пор, пока Вы не выключите питание.
Естественно, что прежде чем составлять программу, необходимо четко определиться как с ее стратегией, так и с принципиальной схемой создаваемого устройства, ведь ***программа составляется не для самой себя, а под конкретное устройство.***

Пример:

Необходимо разработать простую программу, реализующую функцию автоколебательного мультивибратора, с одним-разъединственным выходом.
Форма сигнала - меандр (отношение периода к длительности импульса =2).
Используется **PIC16F84A**.
Под этот выход, можно назначить любой из выводов порта А или В.
Пусть это будет, например, вывод **RB0** порта В.
Следовательно, необходимо установить режим работы вывода **RB0** → "на выход".
Кроме того, что полный цикл программы должен исполняться циклически (периодически), внутри полного цикла программы, должны иметь место быть еще и внутренние циклы.
А как же иначе? Ведь это определяется самим принципом работы автоколебательного мультивибратора, формирующего, в пределах периода, двухуровневый сигнал.
В одном случае, рабочая точка программы должна "закольцеваться" ("задержаться, наматывая витки"), на какое-то время, в подпрограмме формирования уровня **0**, а в другом

случае, она должна "закольцеваться", на какое-то время, в подпрограмме формирования уровня **1**.
Потом опять **0**, потом опять **1** и т.д ....
И так, до "бесконечности" (пока включено питание).
Форма сигнала "меандр" предполагает одинаковое время формирования этих уровней.
Предположим, что для каждого из уровней, это время должно быть равным **100 мкс.**
"Выстраиваем упрощенный скелет" программы:
После команды **Start**, переводим вывод **RB0** на работу "на выход".
Далее, на выводе **RB0**, в течение **100 мкс**., формируем **нулевой уровень**.
Далее, на выводе **RB0**, в течение **100 мкс.**, формируем **единичный уровень**.
Переходим на новый, полный цикл программы, то есть, на команду **Start**, и так до "бесконечности" (пока питание не будет выключено).

Как составляется эта программа?

Текст программы выглядит так:

```
;*****************************************************************************
; Multi.asm           Автоколебательный  мультивибратор.
; PIC16F84A      Кварц 4 мГц.
;=============================================================================
            LIST        p=16F84A    ; Установка типа микроконтроллера.
            __CONFIG    03FF1H      ; Бит защиты выключен, WDT выключен,
                                    ; стандартный XT - генератор.
;=============================================================================
; Определение  положения  регистров  специального  назначения.
;=============================================================================
Status      equ         03h         ; Регистр выбора банка.
TrisB       equ         06h         ; Регистр выбора направления работы выводов
                                    ; порта B.
PortB       equ         06h         ; Регистр управления защелками порта B.
;=============================================================================
; Определение  названия  и  положения  регистров  общего  назначения.
;=============================================================================
Sec         equ         0Ch         ; Счетчик времени полупериода.
;=============================================================================
            org         0           ; Начать выполнение программы с адреса 0 PC.
            goto        Start       ; Переход в ПП Start.


;*****************************************************************************
;                           Текст рабочей части программы.
;*****************************************************************************
; Установка направления работы RB0 - на выход.
;-----------------------------------------------------------------------------
Start       bsf         Status,5    ; Перейти в 1-й банк (установить в 1 5-й бит
                                    ; регистра Status).
            movlw       .0          ; Записать константу 0 в аккумулятор (W).
            movwf       TrisB       ; Скопировать 0 из W в регистр TrisB.

            bcf         Status,5    ; Перейти в 0-й банк (установить в 0 5-й бит
                                    ; регистра Status).
;-----------------------------------------------------------------------------
; Определение времени полупериода (закладка константы в регистр Sec).
;-----------------------------------------------------------------------------
            movlw       .32         ; Записать в регистр W константу .32
            movwf       Sec         ; Скопировать .32 из W в регистр Sec.
;-----------------------------------------------------------------------------
; Формирование на выводе RB0 нулевого уровня.
;-----------------------------------------------------------------------------
            bcf         PortB,0     ; Установить на выходе защелки RB0 ноль.

            nop                     ; Калибровочный машинный цикл.
```

```
            nop                         ; -------------"-------------
Pause_1     decfsz      Sec,F           ; Декремент содержимого регистра Sec с
            goto        Pause_1         ; помещением результата в этот же регистр.
                                        ; Если этот результат не=0, то
                                        ; осуществляется переход
                                        ; в ПП Pause_1 ("закольцовка" в этой ПП).
                                        ; Если =0, то программа исполняется далее.
;-------------------------------------------------------------------------------
; Определение времени полупериода (закладка константы в регистр Sec).
;-------------------------------------------------------------------------------
            movlw       .30             ; Записать в регистр W константу .30
            movwf       Sec             ; Скопировать .30 из W в регистр Sec.
;-------------------------------------------------------------------------------
; Формирование на выводе RB0 единичного уровня.
;-------------------------------------------------------------------------------
            bsf         PortB,0         ; Установить на выходе защелки RB0 единицу.

            nop                         ; То же самое, что и для нулевого уровня,
            nop                         ; только "закольцовка" происходит в ПП Pause_2.
Pause_2     decfsz      Sec,F           ; ------------------"------------------
            goto        Pause_2         ; ------------------"------------------

            goto        Start           ; Переход на новый полный цикл программы.
;*******************************************************************************
            end                         ; Директива конца программы (всегда
                                        ; последняя снизу).
```

Выше текста программы, которая будет исполняться, расположена так называемая **"шапка" программы**.
Она нужна для того, чтобы можно было создать "прошивку" (HEX-файл).
Если этой "шапки" не будет, то при ассемблировании, **MPLAB** "уйдет в решительный отказ".
В "шапке" программы указывается тип ПИКа (в данном случае, PIC16F84A) и значения битов конфигурации (что переключают эти биты → посмотрите в распечатке битов конфигурации).
Примечание: "шапка" программы - моя фантазия, которая облегчает процесс объяснения, так как шапка, она и есть шапка (то, что на самом верху).
Устанавливаем биты конфигурации: **бит защиты CP выключен, сторожевой таймер WDT выключен, кварцевый генератор** (используем кварц на 4мГц.) **работает в режиме стандартного генератора** (XT).
"Разборки" с битами конфигурации будут позднее.
И тип ПИКа, и установленные биты конфигурации (а также "прописка" в программе регистров специального и общего назначения, разрешение на начало выполнения программы и ориентир конца программы - см. ниже) "вводятся в эксплуатацию" так называемыми **директивами** (не путать с командами программы).
О них будет рассказано ниже, а также и в других разделах "Самоучителя..." ("хорошего понемножку". Сейчас "перенапряг" совсем не нужен).
Далее, в "шапке" программы "прописываются" названия и адреса регистров специального назначения (см. область оперативной памяти).
Так как нужно только определить направление работы вывода порта B и далее работать только с этим выводом, то необходимо "прописать" названия и адреса регистров **TRISB** (выбор направления работы выводов порта B), **PORTB** (работа с защелками порта B) и **STATUS** (необходимо переключение банков, так как регистр **TRISB** находится в 1-м банке).
Остальные регистры специального назначения "прописывать" не нужно, так как в программе они не задействованы (обращений к ним нет).
Таким образом, в "шапке" программы, "прописываются" только те регистры специального назначения, которые задействуются в работе программы.
Далее, в "шапке" программы, "прописываются" адреса регистров общего назначения.
С ними нужно определиться.
В предыдущем разделе, речь шла о задержках. Это как раз то, что нужно, так как в данном случае, необходима задержка на **100 мкс**.
Для обеспечения такой малой задержки, достаточно одного регистра общего назначения.
Этот регистр будет участвовать и в процессе формирования нулевого уровня, и в процессе формирования единичного уровня (я называю это совмещением функций).

Назовем его, например, **Sec,** и с помощью директивы **equ**, назначим ему адрес в области оперативной памяти. Например, **0Ch**
Всё. Регистр "прописан".
В конце "шапки", располагается директива, определяющая, с какого именно адреса памяти программ (**PC**) начать исполнение программы (обычно, **org 0** С нулевого адреса **PC**) и команда перехода на начало исполнения программы (обычно, **goto Start**).
Так как команда **goto Start** расположена сразу же после директивы **org 0** (у команды **goto Start** нулевой адрес в памяти программ), то с этой команды и начнется исполнение программы.
Для более сложных программ, "шапка", конечно же, более "объемна", и она содержит в себе большее количество "прописанных" регистров и директив (в том числе и других типов).
В "технологии" оформления "шапки" программы, ничего особо сложного нет.
Нужно только знать, что именно "прописывать", то есть, нужно определиться с теми регистрами, которые будут задействованы по ходу исполнения программы.
Если содержимое "шапки" программы сейчас "мутно" (что вполне естественно), то ничего страшного: далее это прояснится.
Просто сейчас нецелесообразно "уходить" в детальное обсуждение вопросов, связанных с "шапкой", так как это может привести к "отклонению в сторону от генерального направления" затеянного мной объяснения.
Итак, приступаем к составлению программы:
После "срабатывания" команды **goto Start** (она находится в конце "шапки" программы), происходит переход на первую команду рабочей части программы (адрес в счетчике команд PC 0001h).
Она помечена словом **Start**, которое можно считать либо названием подпрограммы, либо меткой - кому как нравится.
Обычно, его считают стандартным названием подпрограммы (слово "старт" всем понятно).
Далее, работаем в соответствии с "выстроенным скелетом" программы.
После инициализации ПИКа (по умолчанию), автоматически устанавливается 0-й банк.
Так как регистр **TrisB** "дислоцируется" в 1-м банке, то переходим в 1-й банк (а иначе, с регистром **TrisB** невозможно будет работать).
Для этого нужно установить 5-й бит регистра **Status** в единицу. Делаем это (**bsf Status,5**).
Так как направления работы выводов порта B **RB1…RB7** нас не интересуют (они не задействованы), то проще всего установить все биты регистра **TrisB** в ноль (все выводы порта B работают "на выход"), то есть, записать в этот регистр константу **0**.
Примечание: по умолчанию, все выводы портов настроены на работу "на вход".
Так как напрямую записать константу в регистр (любой) нельзя, то "транспортировка" этого нуля, в регистр **TrisB**, производится через аккумулятор (через регистр **W**), что Вы и видите в тексте программы (**movlw .0** и **movwf TrisB**).
В этом случае, задействуются байт-ориентированные команды.
Другой способ: задействуем бит-ориентированную команду **BCF**.
В этом случае, в ноль устанавливаются не все биты регистра **TrisB**, а только бит с номером ноль: **bcf TrisB,0** (сброс нулевого бита регистра **TrisB** в **0**).
На практике, чаще применяется первый вариант, так как в тех случаях, когда нужно перенастроить направления работы более 2-х выводов портов (такие случаи часты), можно обойтись минимальным количеством команд.
Всё. Дело сделано. **Теперь вывод RB0 работает "на выход".**
То есть, к нему подключился выход "своей" защелки.
Так как дальнейшие "манипуляции" будут производиться в нулевом банке, то возвращаемся в нулевой банк (устанавливаем **5**-й бит регистра **Status** в **0**: **bcf Status,5**).
Все перечисленные выше операции, можно считать подготовительными.
То есть, подпрограмма **Start** начинается именно с таких операций.
А как же иначе? Ведь к "боевым действиям" нужно подготовиться.
Для того чтобы осуществить декремент (или инкремент) содержимого регистра **Sec** (вспомните про задержки), необходимо сначала определиться с его содержимым (какую "начальную точку отсчета" задавать?).
Предположим, что с этим содержимым мы определились: в регистр **Sec** записывается число **.32** (почему именно оно → см. далее).
Посмотрите в текст программы.

Число **.32** (точка - атрибут десятичной системы исчисления) является константой (то, что задается программистом).
"Напрямую" записать это число в регистр **Sec** (и вообще, в любой регистр) нельзя.
Эта операция осуществляется через "посредника" в виде регистра **W**
(**movlw .32** и **movwf Sec**).
Число **.32** в регистр **Sec** записано, и в дальнейшем, содержимое этого регистра (то есть, число) можно последовательно декрементировать (уменьшать на 1).
Итак, все готово для управления защелкой **RB0** порта B.
Формируем нулевой уровень на выводе **RB0**.
Применяем бит - ориентированную команду **BCF** и указываем в ней нулевой бит регистра **PortB**, который нужно сбросить в **0** (**bcf PortB,0**).
Все. Дело сделано, и на выводе **RB0,** ноль зафиксируется до тех пор, пока мы его не заменим на **1** (вспомните о том, что защелка является банальным триггером, то есть, элементом оперативной памяти).
Теперь нужно сделать так, чтобы с момента установки, на выводе **RB0**, нуля и до момента смены этого нуля на единицу, прошло ровно **100 мкс.**
Предположим, что с учетом того, что в регистр **Sec**, ранее была записана "нужная" константа (**.32**), этот интервал времени сформирован.
Установка, на выводе **RB0,** единицы происходит так же, как и установка нуля, только применяется команда **BSF** (**bsf PortB,0**).
Точно таким же образом, как описано выше, для формирования, на выводе **RB0**, единичного уровня, происходит запись, в регистр **Sec**, следующей константы.
Давайте разберемся, почему числовое значение константы, обеспечивающей формирование нулевого уровня, должно быть не абы какое, а **.32** ?
Применяется кварц на 4мГц., следовательно, один машинный цикл равен 1мкс.
Таким образом, с момента установки, на выводе **RB0**, нуля и до смены его на единицу должно пройти 100 машинных циклов.
Подпрограмма задержки **Pause_1** - циклическая.
Если результат декремента содержимого регистра **Sec** не равен нулю, то один "виток" этой подпрограммы будет отработан за 3 машинных цикла: команда **DECFSZ** выполняется за один м.ц., а команда **GOTO** - за два.
Таким образом, уменьшение содержимого регистра **Sec** на единицу происходит за 3 м.ц. (3мкс.).
Если результат декремента содержимого регистра **Sec** равен нулю, то команда **GOTO** не исполняется (вместо нее, "виртуальный" NOP), и выход из подпрограммы **Pause_1** происходит за 2 м.ц.
Следовательно, константе .32 соответствует 32x3-1=95 м.ц. (95мкс.).
К 95-ти, плюсуем 2 м.ц. двух команд **NOP**, 2 м.ц. команд установки константы в регистр **Sec** для формирования времени единичного уровня на выводе **RB0** (**movlw .30** и **movwf Sec**) и 1 м.ц. команды установки, на выводе **RB0**, единицы (**bsf PortB,0**).
Получаем ровно **100 мкс.**
Теперь о двух **NOP**ах, которые стоят перед ПП **Pause_1**.
Если сделать константу равной **.33**-м, то получится неустранимый "перебор" (рабочие команды из программы не выкинешь).
Если константа равна **.32**-м, то получается устранимый "недобор" в 2 мкс., который устраняется добавлением двух **NOP**ов (дополнительная задержка на 2 мкс.).
После того, как результат декремента содержимого регистра **Sec** станет равным нулю, рабочая точка программы выйдет из этой "закольцовки" по сценарию **"программа исполняется далее"**.
После этого, с целью обеспечения условий для дальнейшего формирования единичного уровня (в течение ста мкс.), в регистр **Sec**, необходимо записать "новую" константу.
Мотивация: "на влёте" в процедуру формирования единичного уровня, в регистре **Sec** "лежит" ноль.
Такая запись и имеет место быть (**movlw .30** и **movwf Sec**).
Различие числовых значений констант (**.32** и **.30**) при формировании нулевого и единичного уровней, объясняется тем, что при формировании, на выводе **RB0**, единичного уровня, исполняется команда **GOTO** (2 м.ц.) и команды ПП **Start** (в конце текста программы, осуществляется безусловный переход в ПП **Start**).
Кто имеет такое желание - посчитайте.

Если такие расчеты пугают, то могу Вас успокоить: в MPLAB, с которой мы будем работать далее, имеется счетчик машинных циклов ("секундомер"), и при отладке циклических подпрограмм, с его помощью, подобного рода трудозатраты существенно снижаются.
Разбираемся с **директивами ассемблера.**
Директивы, в отличие от команд, не включаются в выходной код.
Выходной код - результат ассемблирования текста программы, а проще говоря, содержимое HEX-файла ("прошивки"), то есть, файла, предназначенного для открытия в программе, обслуживающей программатор.
Тогда зачем вообще нужны директивы?
Дело в том, что директивы, за "пределы" MPLAB (интегрированной среды разработки для ПИКов), не выходят и исполняются только внутри MPLAB.
Те директивы, которые предназначены для применения в рабочей части программы (с учетом рабочей части директивы), имеют свои так называемые "разложения на команды".
То есть, одна такая директива как бы заменяет конкретную группу команд, которая автоматически (в MPLAB) ставится ей в соответствие.
Если в тексте рабочей части программы имеется такая директива ("элемент удобства" при составлении текста программы), то исполняется не она, а ее разложение на команды.
Точно так же программа исполнится, если в тексте рабочей части программы заменить директиву на ее разложение на команды.
Если директива находится вне "границ" рабочей части программы (в "шапке" или директива **END**), то естественно, что никакого ее разложения на команды не будет.
Вне рабочей части программы, до команды перехода на начало исполнения рабочей части программы (**goto Start**), MPLAB "понимает" директивы и "не понимает" команд.
Посмотрите в "шапку" программы **Multi.asm:** до команды перехода на начало исполнения рабочей части программы, Вы не найдете ни одной команды.
Это "внутреннее дело" MPLAB (тайна сия велика есть).
Смысл же состоит в том, что директивы, отрабатываемые вне рабочей части программы, либо обеспечивают подготовку ее исполнения, либо "обозначают ее границы".
Результаты всей этой "свистопляски закладываются в прошивку", при ее создании.
Что касается директив рабочей части программы, то они, по существу, - элемент "оформительского" удобства, так как в случаях их применения, в тексте программы не нужно "расписывать" соответствующие им группы команд (разложения).
MPLAB, без вмешательства программиста, "воткнет" это разложение в то место памяти программ, в которое его и нужно "воткнуть" и создаст соответствующий HEX файл ("прошивку").
Директивы можно уподобить администратору - распорядителю какого-то мероприятия.
Задача этого администратора: перед началом "мероприятия" (рабочей части программы), для того чтобы оно прошло успешно, "расставить всё по своим местам", а также и в "меру своих возможностей, поучаствовать в этом мероприятии".
Не смотря на большое количество всевозможных директив, на практике применяется ограниченное их количество.
Некоторые директивы:

Директивы макроассемблера MPASM

1. **2.**	**CBLOCK** **ENDC**	**Определение блока констант.** **Используется для размещения нескольких констант в памяти программ и памяти данных.** **Сначала указывается стартовый адрес для первой константы, последующие адреса декрементируются.** **Список заканчивается директивой ENDC.**	**cblock 0x20** **nameA,nameB ;адрес20,21** **nameC,nameD ;адрес22,23** **endc**
3.	**END**	**Окончание программы (конец всех команд)**	**……** **……** **end**
		Присваивает неизменное значение	

4. 5.	EQU SET	константе. Присваивает значение константе, которое можно переопределить.	set nameA equ 0x05 ;присво- ;ить константе nameA 0x05
5.	INCLUDE	Подключение дополнительного исходного файла	#include p16f84a.inc ;подклю- ;чение файла с описаниями ;регистров спец. назначения
6.	_ _CONFIG	Установка битов конфигурации	См. таблицу символов конфигурации
7. 8.	CONSTANT VARIABLE	Определение неизменной символьной константы Определение символьной константы, значение которой в последствии можно переопределить.	*constant cnt=255* variable temp=0xF0 constant cnt1=cnt2+cnt3
9.	ORG	Установить начальный адрес программы. При отсутствии ORG программа начинается с нулевого адреса.	*<метка1> org 0x20* ;вектор с адресом 20 <метка2> org <метка1>+0x10 ;вектор с адресом 30
10.	RADIX	Система исчисления по умолчанию (hex – 16, dec – 10, oct – 8)	*radix dec*

Обратите внимание на директивы **EQU, __CONFIG, ORG, END**. Они используются практически во всех программах (примечание: перед директивой **CONFIG** не одно, а 2 подчеркивания, идущих подряд.)
Есть они и в программе **Multi.asm**.
Найдите их в тексте этой программы, и Вы поймете что они "администрируют" (распределяют, назначают и ограничивают).
Также обратите внимание на то, что с помощью директивы **EQU,** регистры **TrisB** и **PortB** "прописаны" по одинаковым адресам (**06h**).
С нулевым банком все понятно: регистр **PortB** имеет в области оперативной памяти адрес **06h**, а вот регистр **TrisB** (первый банк) "прописан" не по адресу **86h**, а все по тому же адресу **06h**.
В чем дело?
В адресах регистров специального назначения, ноль (устанавливается по умолчанию) меняется на восьмерку только тогда, когда программно осуществлен переход в 1-й банк.
Без учета этого, нумерация адресов регистров специального назначения обеих банков одинакова.
А раз это так, то регистр **TrisB** можно прописать не по адресу **86h**, а по адресу **06h**.
После исполнения команды выбора 1-го банка, **06h** автоматически "превращается" в **86h**, что и нужно.
Такой способ "прописки" удобен тем, что в окне результата ассемблирования, будут отсутствовать сообщения информативного характера типа "сие лежит не в нулевом банке", что на мой взгляд, удобно.
Если прописать регистр **TrisB** по адресу 86h, то такие сообщения будут выдаваться.
Они не являются ошибками, но нервируют и отвлекают внимание.
Особенно на первых порах.
Рекомендую Вам, при "прописке" адресов регистров специального назначения 1-го банка, обозначенных, в области оперативной памяти, **черным цветом**, назначать им адреса по принципу типа **"минус 80"**, то есть, так, как сделано в программе **Multi.asm** (в части касающейся "прописки" регистра **TrisB**).
В этом случае, сообщение об успешном ассемблировании будет самым коротким (4 строки).
Но можно "прописать" и по "штатным" адресам (будут выдаваться сообщения информативного характера). "Это на любителя".
В любом из этих случаев, ошибок не будет.

Синтаксические правила написания программы.

MPLAB "замучает" Вас сообщениями об ошибках, если не выполнены определенные правила написания текста программы.
Давайте в них первично разберёмся, на примере программы **Multi.asm**.
Все, что находится правее точки с запятой (**;**), MPLAB "не видит".
То есть, после точки с запятой, программист может "настукать" всё, что ему "в голову взбредёт" (по своему усмотрению и на любом языке).
В части касающейся этого, все просто, как мычание коровы. А вот дальше - сложнее.
Активная часть и "шапки" программы, и рабочей части программы, разделена на 3 столбца.
Левые буквы содержимого 1-го (левого) столбца должны занимать крайние левые позиции строк, на которых они находятся. Это, как бы, "начальная точка отсчета".
2-й и 3-й столбцы находятся правее, и первые символы их содержимого должны (в идеале) располагаться на одной вертикальной линии.
Последние символы содержимого столбца не должны "наезжать" на первые символы соседнего справа столбца. Между ними должен быть хотя бы один пробел.
Чтобы "не забивать Вам голову" рассуждениями о возможных компоновках расстояний между столбцами, рекомендую пользоваться следующим "правилом 12-ти пробелов":

1. ***От крайней левой позиции строки (начало 1-го столбца) отсчитываете 12 пробелов. Это будет начало 2-го столбца.***
2. ***Затем отсчитываете 12 пробелов от начала 2-го столбца. Это будет начало 3-го столбца.***
3. ***Затем отсчитываете 12 пробелов от начала 3-го столбца. Это будет начало комментариев.***

Такой вариант "проверен жизнью". Я пользуюсь им всегда и Вам советую.

Содержимое столбцов.

"Шапка" программы.
В 1-м столбце (левом) ***располагаются названия регистров и названия битов*** (в программе **Multi.asm** их нет, но в других, более сложных программах, они есть).
Во 2-м столбце находятся директивы и команда перехода на начало исполнения программы.
В 3-м столбце находятся рабочие части директив и название подпрограммы, с которой начинается исполнение программы (в данном случае, **Start**).
Для регистров, определяется их адрес.
Для битов, определяется их номер (в пределах байта).
После этого, в рабочей части программы, при обращении команд к этим битам, можно, вместо номера бита, указать присвоенное ему название (обычно, стандартное, но можно назначить и свое).
В конце "шапки" программы, располагается "связка" **org 0** и **goto Start** (начать исполнение программы с ПП **Start**, первая команда которой имеет нулевой адрес), а при наличии в программе подпрограммы прерываний - директива **org 4** (об этом - позже).
Рабочая часть программы.
В 1-м столбце располагаются названия подпрограмм и меток .
Во 2-м столбце располагаются команды и директивы.
В 3-м столбце располагаются рабочие части команд и директив.
В конце программы, обязательно должна присутствовать директива **END** {что-то типа "сюда (вниз) не ходи. Ходи туда (вверх)"}.
Если убрать все то, что находится правее точек с запятой (оформление, комментарии), то это и будет текст программы в "чистом виде", то есть, то, с чем работает MPLAB при ассемблировании (создании HEX - файла).

Теперь - **о форме представления чисел**.

Чтобы MPLAB "поняла", в какой системе исчисления представлено число, необходимо его соответствующим образом оформить:

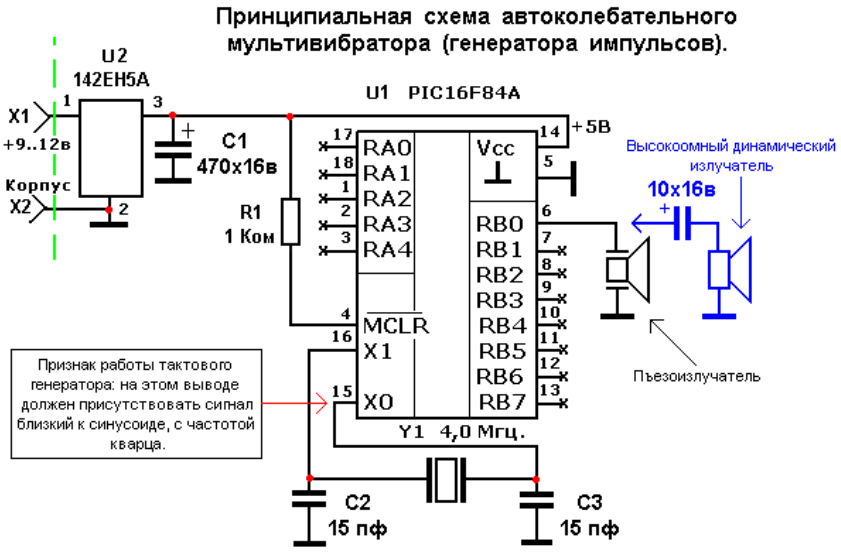
СИНТАКСИС ЧИСЛОВЫХ ЗНАЧЕНИЙ В РАЗЛИЧНЫХ СИСТЕМАХ ИСЧИСЛЕНИЯ

Формат	Синтаксис	Примеры
16 - ричный	**H´число´** **0xчисло**	**H´9f´** **0x9f**
10 - тичный	**D´число´** **.число**	**D´100´** **.100**
8 - ричный	**O´число´**	**O´777´**

2 - ичный	В´число´	В´00001111´
Символьный (ASCII)	´символ´ А´символ´	´С´ А´С´

На мой взгляд, в каких-то особых комментариях эта таблица не нуждается.
Для того чтобы, на первых порах, не создавать путаницы, связанной с переводом чисел из одной системы исчисления в другую, в дальнейшем, я буду, преимущественно, пользоваться привычной, то есть - десятичной системой исчисления.
В программе **Multi.asm,** константы представлены именно в десятичной системе исчисления (с точкой слева).
До девятки включительно, 16-ричная и 10-тичная системы исчисления одинаковы.
В этом случае (число не более 9-ти), все равно, в какой системе исчисления, в 16-ричной или в 10-тичной, происходит отображение числа.
То есть, в этом случае, число можно "прописать" в привычном (удобном), десятичном виде, без указания атрибутов системы исчисления.
Если в тексте программы "прописано" число, не "укомплектованное" атрибутом системы исчисления, то по умолчанию, это число будет "воспринято" MPLABом как 16-ричное.
Если это число не более 9-ти, то какая разница, ведь в 10-чном виде оно точно такое же, как и в 16-ричном.
После девятки, "этот номер уже не пройдет", так как появляются расхождения.
Бинарная система исчисления стоИт, как бы, "особняком" и этим правилам не подчиняется.
8-ричная система исчисления используется очень редко.
Символьная система исчисления "привязана" к стандартному, международному коду **ASCII**
В "шапке" программы, адреса регистров лучше указывать в 16-ричном виде, а всё остальное – на выбор. Как кому удобнее. В зависимости от конкретных обстоятельств.
При указании номера бита, по причине того, что номера битов не являются числами, большими чем 9, просто пишется номер бита (без атрибутов системы исчисления).
Итак, правила оформления текста программы, на самом деле, не являются чем-то страшно трудным. Наоборот, они достаточно просты и освоить их не составляет великого труда.
Промежуточный итог: мы сейчас уже вплотную подошли к практической работе с текстами программ, что предполагает "ввод в действие тяжелой артиллерии" в виде "набора" программ интегрированной среды разработки для ПИКов с названием **MPLAB**.

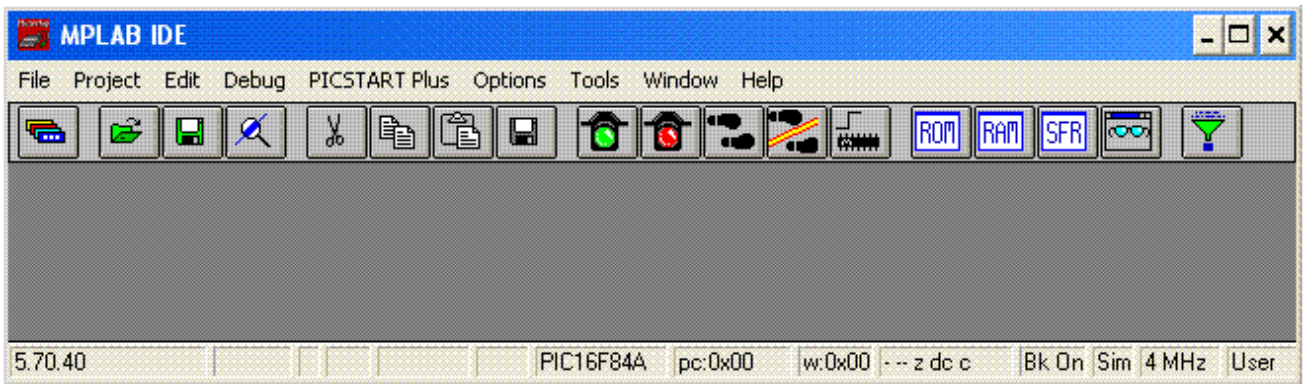
Приложение
Принципиальная схема автоколебательного мультивибратора (генератора импульсов)



Синим цветом показан случай подключения высокоомного, динамического излучателя (например, от телефонного аппарата или наушников).
Во избежание перегрузки выходного каскада защелки, подключать низкоомный, динамический излучатель (громкоговоритель) не стоит.
Лучший вариант - подключение пьезоэлектрического излучателя.

5. Интегрированная среда проектирования MPLAB IDE и работа в ней.

MPLAB это интегрированная среда проектирования для ПИКов, то есть, набор программ, которые позволяют программисту, с максимальным комфортом, составлять, отлаживать и оптимизировать текст программы, а также создавать (после этого) **HEX - файл программы** ("прошивку").

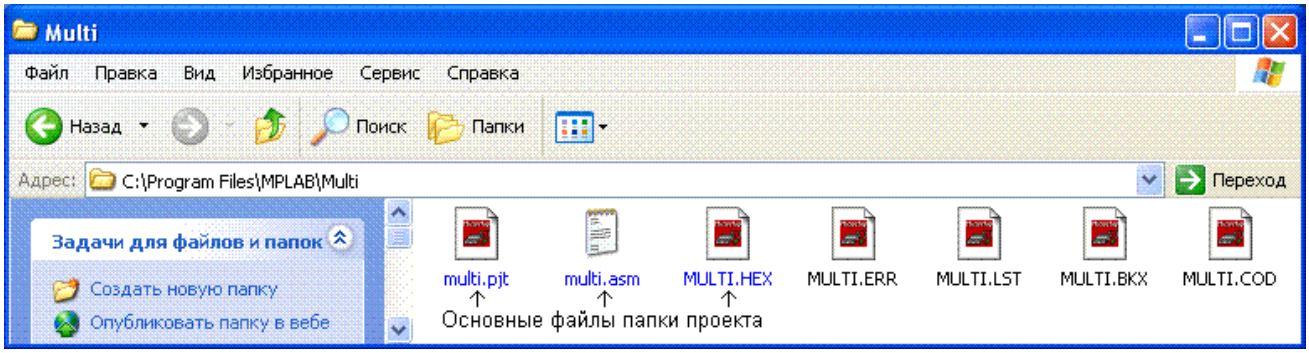


Она также имеет и другие "навороты" (в самом уважительном смысле этого слова), которыми начинающим, для их же пользы, не стоит, пока, "забивать себе голову".
Давайте сначала разберемся с "прожиточным минимумом".
Предполагается, что **MPLAB** уже установлен на Вашем компьютера и готов к работе.

Общие положения.

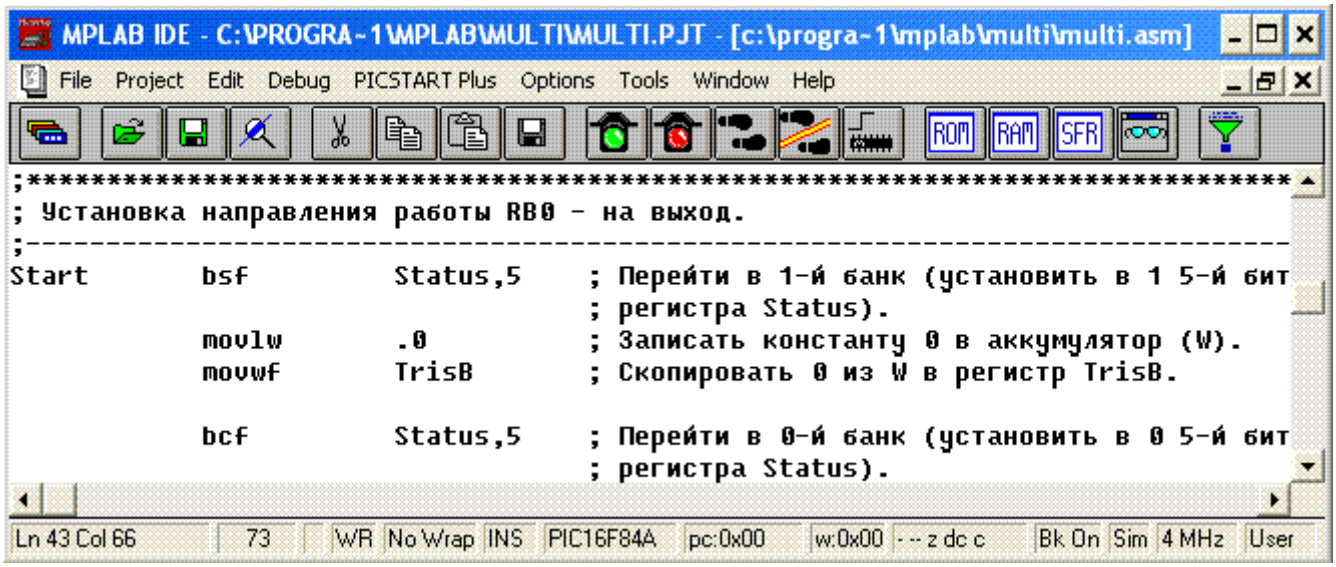
MPLAB создана конкретно для ПИКов.
Для микроконтроллеров других видов имеются свои среды проектирования.
MPLAB поддерживает все типы ПИКов, за исключением самых новейших разработок (их можно догрузить) и существенно облегчает работу программиста за счет совмещения нескольких, необходимых для работы программиста, функций.
MPLAB можно представить себе как надежного друга, который берет на себя значительную часть рутинной и трудоемкой работы.
Этот друг укажет на ошибку, если она совершена, и поможет Вам в решении проблемных задач, которых, при создании и отладке программ, Вы встретите множество.
Редко когда бывает так, что составление программы идет "как по маслу".
Скорее всего такое - исключение из правил (особенно, для достаточно сложных программ), и поэтому "прожиточный минимум" **MPLAB** нужно знать как "Отче наше".
Сразу же нужно уяснить следующее: работа по созданию текста своей программы или работа по изменению текста чужой программы всегда начинается с создания так называемого **проекта**.
Что-то изменять в тексте программы, производить ее оптимизацию и отладку лучше всего в проекте.
В этом случае, с максимальным комфортом, можно пользоваться полным набором возможностей **MPLAB**.
Многое можно сделать и вне проекта, но в этом случае, возникают определенные неудобства, которые нам ни к чему.
В целях определения ясных и понятных "правил игры", давайте договоримся о следующем: составлять, оптимизировать, отлаживать тексты программ, подпрограмм, и вообще, вносить в их тексты какие-либо изменения, мы будем только в **проекте**.
Вне проекта, мы будем только просматривать тексты программ и HEX – файлы, а также открывать файлы, предназначенные для копирования в проект.
В пределах проекта, осуществляется работа только со "своими" текстами программ.
Для того чтобы изменить их содержимое, необходимо либо сделать это "вручную", либо скопировать в них (полностью или частично), через буфер обмена, содержимое какой-то другой программы.
Вне проекта, никаких изменений в тексты программ мы вносить не будем (хотя и можно).
Советую Вам и в дальнейшем, работать по этим правилам.
На мой взгляд, это самый оптимальный и безпроблемный вариант.
В проекте, программист может создавать "свой" текст программы или/и вносить в него

изменения, а также вносить изменения в текст "чужой" программы (**ASM-файл**).
Практической необходимости в "ручном" изменении содержимого **HEX-файла** ("прошивки"), в подавляющем большинстве случаев, нет, так как проще и эффективнее работать не с "прошивкой", а с текстом программы.
Если нет ошибок, то в процессе любого ассемблирования текста программы, **MPLAB** автоматически создает **HEX-файл** программы (мозги отдыхают).
В ходе работы над текстом программы, это можно делать многократно (хоть миллион раз), не боясь того, что при этом "наплодится" большое количество **HEX-файлов**, в которых можно запутаться.
Объяснение этому простое: по "правилам игры" **MPLAB**а, "право на жизнь" имеет только один **HEX-файл.** Тот, который создан при последнем ассемблировании.
Он просто записывается "по верху предшественника" ("предшественник усоп").
Программисту не нужно знать все детали процесса ассемблирования.
MPLAB "все сделает в лучшем виде".
Самое главное - было бы что ассемблировать, и чтобы это "что" было без ошибок, а иначе **MPLAB** "выдаст на гора" сообщение об ошибках, с указанием вида ошибок, мест ошибок в тексте программы и "откажется" создавать HEX-файл.
Если текст программы (любой) открыт в **MPLAB** и нужно создать HEX-файл, то программисту нужно только "запустить" процесс ассемблирования и дождаться его окончания.
Если происходит работа в проекте, то HEX-файл, "в автомате, уходит" в папку проекта (см. ниже).
Кроме HEX-файла, **MPLAB** создает еще несколько файлов с различными расширениями (ASM-файл не в счет), на которые можно "закрыть глаза".
Лично я, после того как заканчиваю работу над программой, либо вообще удаляю их из папки проекта, либо просто не обращаю на них внимания.
Например, содержимое папки проекта с названием **Multi**, после окончания работы над проектом, выглядит так:



Основные файлы папки проекта, без которых обойтись нельзя:
1.Файл проекта с расширением .PJT: чисто "технологический" файл, который "рулит" всеми остальными файлами папки.
Он создается при создании проекта. К нему можно применить принцип: "создал и забыл".
Он "живет по законам" **MPLAB** и нам там делать нечего.
2. Файл текста программы с расширением .ASM.
3. Результат ассемблирования текста программы - **файл с расширением .HEX**.

Сразу после создания проекта, в папке проекта, имеется **PJT-файл** и "пустой**" ASM-файл**.
Так как ассемблирования не производилось (да и что ассемблировать, ведь **ASM-файл** "пустой"), то **HEX-файл** будет отсутствовать, но он появится, в этой папке, после первого же успешного ассемблирования.
В "пустом" **ASM-файле** можно либо "настучать" программу с "нуля", либо скопировать в него (через буфер обмена) предварительно заготовленные части будущей программы и далее ее доработать, либо скопировать в него текст "чужой" программы, с целью его изменения, либо в комплексе.
Все манипуляции с содержимым **ASM-файла** происходят в специальном текстовом редакторе **MPLAB**.



По принципу своей работы, он мало чем отличается от других текстовых редакторов, и работать с ним проще, чем, например, с редактором "Винворда".
Специальным он называется по той причине, что "встроен" в **MPLAB**.
Что дает такая "встроенность"?
При наличии ошибок в тексте программы, после щелчка (в окне результата ассемблирования) по той или иной строке сообщения об ошибке, в редакторе выделяется строка, в которой допущена ошибка.
В тексте программы может быть организован поиск чего-то повторяющегося, например, команд, которые обращаются к тому или иному регистру.
В ходе отладки программы, Вы будете видеть исполняемую команду (она выделяется).
Для определения времени исполнения составных частей программы, можно назначить точки остановки и задействовать секундомер. И т.д.
Предполагаю, что для Вас еще не совсем понятны эти нюансы. Я говорю о них только лишь для того, чтобы создать общее представление о том, с чем придется иметь дело.
Ничего страшного. На практике все гораздо проще, чем на словах.
Скоро Вы убедитесь в этом сами.
Итак, текстовый редактор это "посредник в общении" программиста и программы, в ходе ее составления, оптимизации и отладки.
Поэтому основную часть времени работы с программой Вы будете проводить именно в нем.
За этим немудреным (с точки зрения манипуляций с текстом) редактором, тем не менее, стоит вся "мощь" **MPLAB** и в этом отношении, любой, даже самый "крутой" редактор, "работающий сам по себе", ему и "в подметки не годится".
Зауважали? Правильно сделали.
Спецы "Микрочипа" свою зарплату отрабатывают "по полной программе".
Совместно с текстовым редактором, работает так называемая программа - **симулятор**.
Уже из названия понятно, что она что-то "симулирует", то есть, **"создает видимость работы"**.
Симулятор является средством отладки программы и позволяет, без применения реального ПИКа, сымитировать работу программы в своего рода "виртуальном" ПИКе, который, специально для этого, создается программой-симулятором.
Этот "виртуальный" ПИК, на самом деле, физически не существует.
Он создается программными средствами **MPLAB**, причем, конкретно для каждого типа ПИКа (в зависимости от того, какой из типов ПИКа выбран).
Симулятор, конечно же, не "всесилен", но с его помощью (плюс некоторые хитрости) можно проверить работу программы в большинстве возможных режимов (как в целом, так и по частям), скорректировать ее временные характеристики, найти ошибки и проверить их устранение.
Симулятор это та "палочка - выручалочка", которая применяется в случае "корявой" работы программы, зависания ("глюка") или в случае, когда программа вообще не работает или работает не так, как нужно.
Программа - симулятор покажет Вам, по каким адресам находятся те или иные регистры и

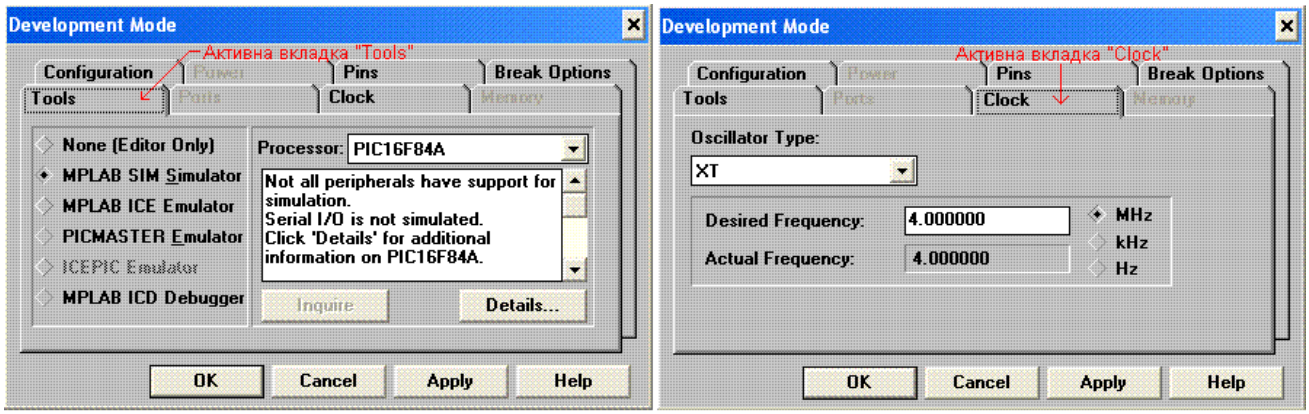
какие изменения происходят с их содержимым по ходу исполнения программы, посчитает количество машинных циклов исполнения того или иного, выбранного участка программы, поможет сымитировать ту или иную ситуацию и т.д.
И все это → без "прошивки" реального ПИКа, что экономит кучу времени и нервов.
По большому счету, программисту вовсе не обязательно знать, с помощью какой именно программы, из набора программ **MPLAB,** производятся те или иные действия.
Достаточно только знать, какое именно действие нужно произвести, как его "запустить" и как проверить результат этого действия, но все-таки будет лучше, если программист, хотя бы в общих чертах, имеет представление о структуре "инструмента", с которым он работает.
Хуже от этого не будет.
Итак, ближе к практике.

Работа с MPLAB

Сейчас, разумнее всего, не создавать программу с "нуля", а создать проект какой-нибудь готовой программы.
В качестве этой программы, я буду использовать, приведенную выше, в качестве примера, программу **Multi.asm**
В дальнейшем, на ее примере, будет показано, как производится ассемблирование, как редактируется текст программы и как происходит отладка программы.

Настройка MPLAB.

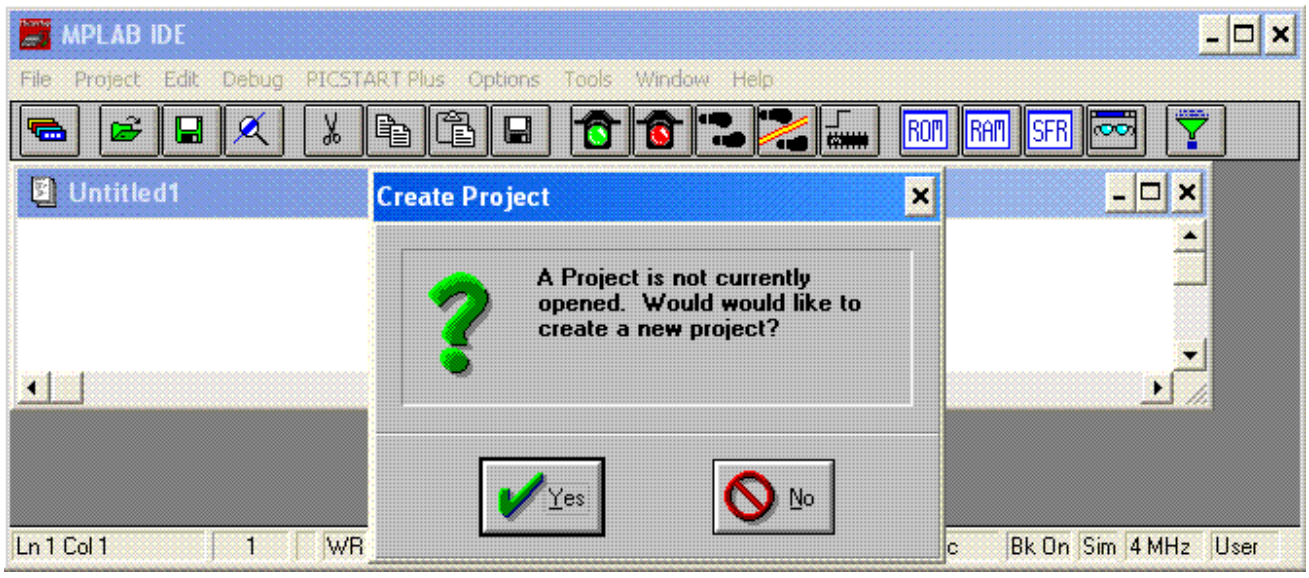
Мудрить с настройкой не нужно.
По умолчанию, разработчики **MPLAB** установили оптимальный вариант настройки, за что им большое спасибо.
Что нужно сделать?
Нужно проконтролировать (на всякий случай) включение симулятора и выставить в ***Опциях*** тот тип ПИКа, который задействован в программе (в нашем случае это **PIC16F84A**), определить стандартный режим работы генератора (**XT**), значение частоты кварца (**4мГц**) и единицу измерения частоты (**мГц**).
Запустите **MPLAB**.
После этого, Вы увидите "фасад" программы с пустым окном программы.
Управление - привычное: сверху - главное меню, под ним - кнопки, дублирующие основные действия.
Самая левая кнопка - выбор различных раскладок этих кнопок.
Таких раскладок - 4 (на все случаи жизни), но лично я, пользуюсь только одной.
Если в этой раскладке не хватает какого-то нужного мне действия, то я использую команды главного меню.
Пощелкайте по самой левой кнопке: найдите и оставьте ту раскладку, в которой кнопки с двумя светофорами занимают 9-ю и 10-ю позиции слева.
В главном меню щелкните по слову ***Options***, а затем - по строке ***Development Mode***.
Откроется окно ***Development Mode*** с активной вкладкой ***Tools***.
Если вкладка ***Tools*** не активна, то щелкните по ней.
На вкладке ***Tools*** должна быть поставлена точка перед ***MPLAB SIM Simulator*** (симулятор включен).
Если это не так, то поставьте ее и не снимайте, так как симулятор нужен всегда.
Щелкните по кнопке списка ***Processor***.
Вы увидите список ПИКов, с которыми работает **MPLAB**.
Нужно выбрать из этого списка **PIC16F84A**.
Перейдите на вкладку ***Clock*** и поставьте там точку перед ***MHz***.
Укажите значение частоты применяемого кварца (***4.000000***).
В выпадающем списке ***Oscillator Type***, нужно выбрать ***XT*** (стандартный генератор).
В конечном итоге, Вы должны наблюдать следующее (это две отдельные картинки вкладок. Я их просто совместил):



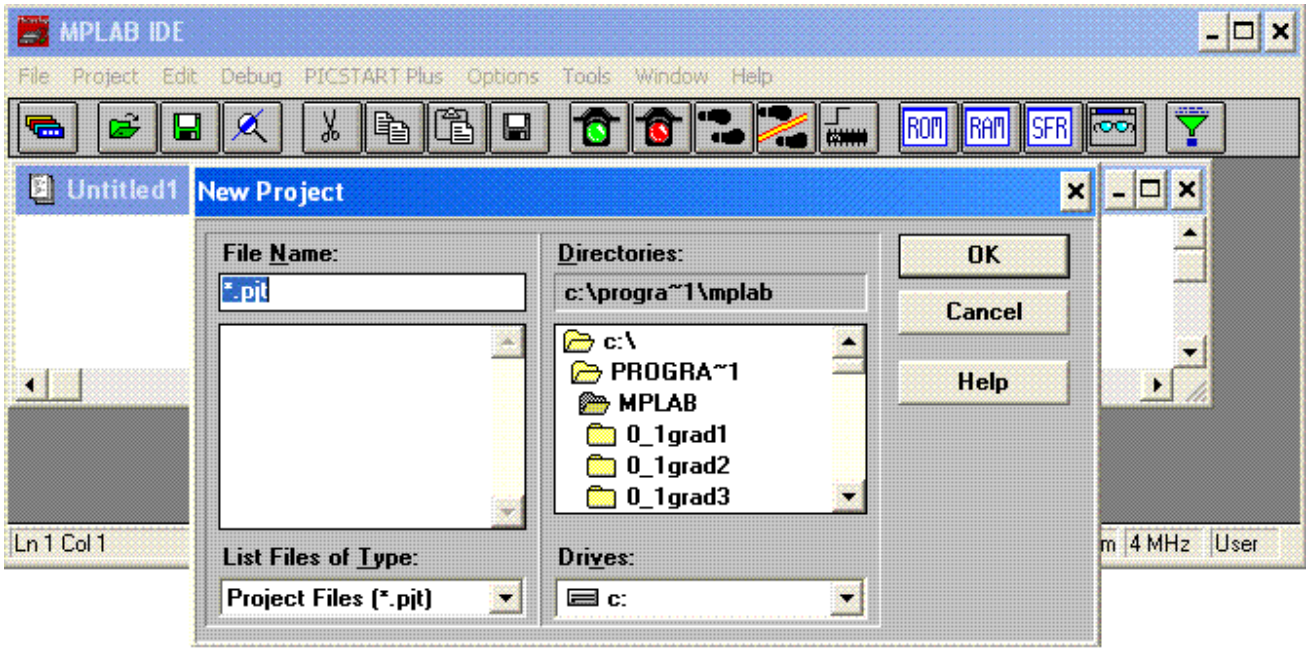
Щелкните по ***OK***.
Настройка опций **MPLAB** закончена.
Остальные настройки - по умолчанию (то есть, делать больше ничего не нужно).

Создаем проект

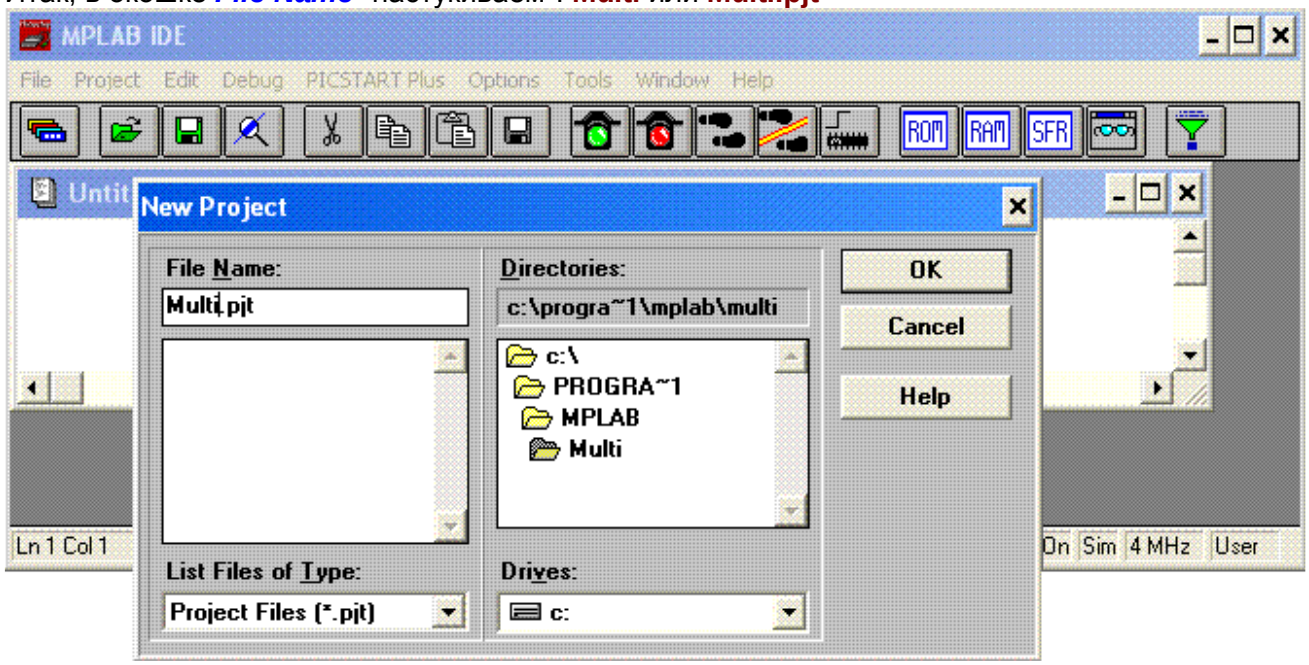
Если для файлов проекта заранее не определено место их хранения, то все они будут "свалены в кучу" внутри **MPLAB**, и Вы будете долго и нудно их искать среди большого количества файлов программы.
В целях недопущения такого "бардака", необходимо заранее (до создания проекта) определить место, где эти файлы будут компактно храниться.
Советую Вам делать так, как делаю я:
Не запуская **MPLAB**, зайдите в папку программы **MPLAB** (например, через проводник).
Для ориентира: Вы должны увидеть папку **Example**.
Здесь же создайте "глобальную" папку с названием, например, **Pic** (для хранения различных проектов), а внутри папки **Pic** создайте пустую папку **Multi** (в нее мы и поместим все файлы проекта **Multi**).
Таким образом, в папке **Pic** будут храниться папки с файлами различных проектов, которые мы будем создавать. Дешево, сердито и никакого "бардака".
Открываем **MPLAB**.
Сначала необходимо создать пустой файл, который, в дальнейшем, мы как-нибудь назовем, присвоим расширение (**.ASM**) и сохраним в папке **Multi**.
Это делается очень просто: щелкните по слову ***File*** в главном меню, а затем по ***New*** в выпадающем списке, после чего откроется пустое окно файла, которому программа присвоит название **Untitled1** (безымянный №1), 2, 3...
Это не имеет значения, так как далее, это название будет изменено.
Одновременно с созданием "пустышки", программа предложит Вам создать **проект**.



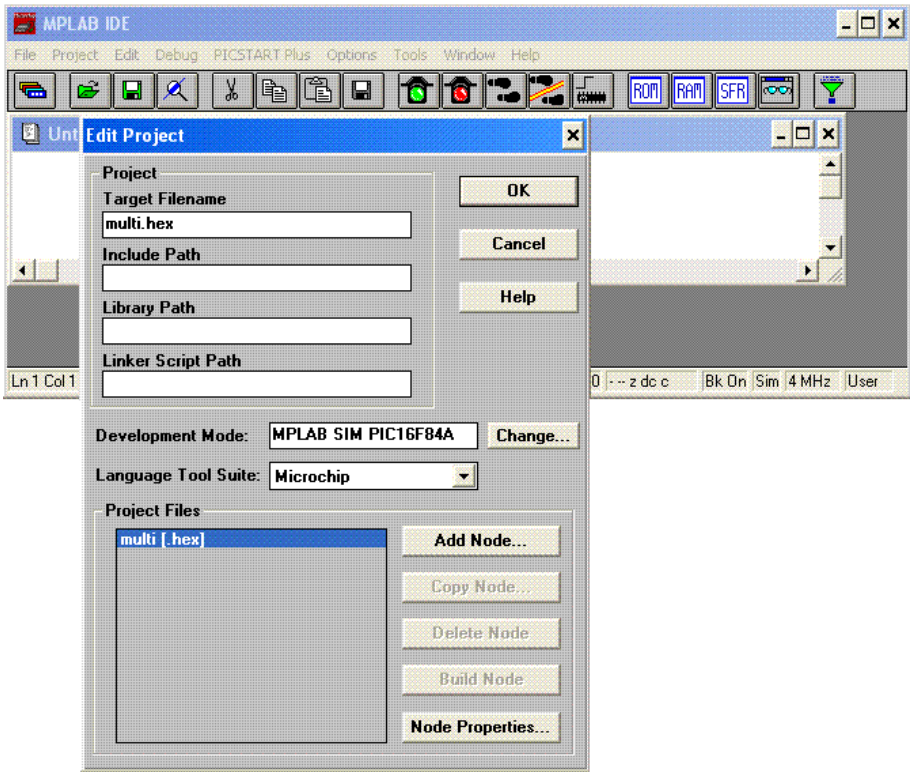
Щелкните по кнопке ***Yes***. Откроется окно ***New Project***.



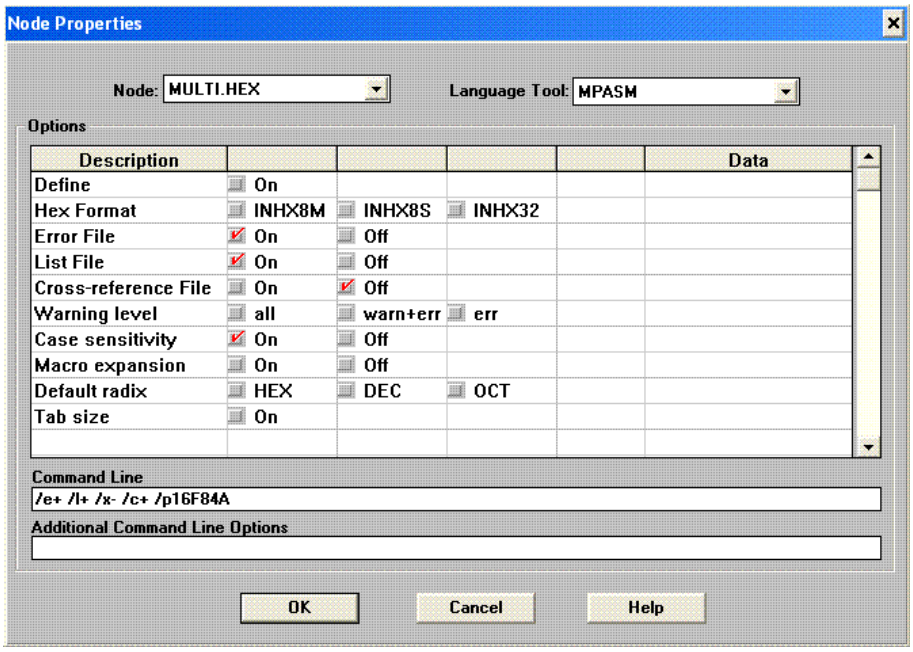
Справа, под словом ***Directories***, Вы увидите что-то типа "недоношенного" Виндовского проводника, а еще ниже - окошко выбора дисков компьютера. Выбираете тот диск, на котором у Вас находится **MPLAB** и в проводнике щелкаете по папке **MPLAB**, после чего Вы увидите ее содержимое. Найдите предварительно созданную папку **Pic**.
Щелкните по ней, а затем по папке **Multi**.
После того, как Вы увидите, что она открылась (в том смысле, что картинка папки изменится с закрытой на открытую), обратите внимание на левую часть окна ***New Project***.
Снизу (***List Files of Type***) Вы увидите, что программа автоматически присвоит создаваемому файлу проекта расширение **.PJT**, поэтому в окошке ***File Name*** можно даже указать имя файла проекта без расширения (это относится и к файлам с расширениями **.ASM** и **.HEX**, о которых речь пойдет далее), а можно и с расширением. Кому как нравится.
Лучше всего присвоить всем файлам проекта одно и то же имя. В нашем случае - **Multi**.
Так как у всех файлов проекта разные расширения, то "конфликтовать" между собой они не будут.
Внимание: при создании проекта, русские буквы применять нельзя. Нужно применять только латинские буквы и цифры. Все названия должны включать в себя не более 8-ми (включительно) символов (без учета символов расширения), а иначе **MPLAB** будет "ругаться".
Итак, в окошке ***File Name*** "настукиваем": **Multi** или **Multi.pjt**



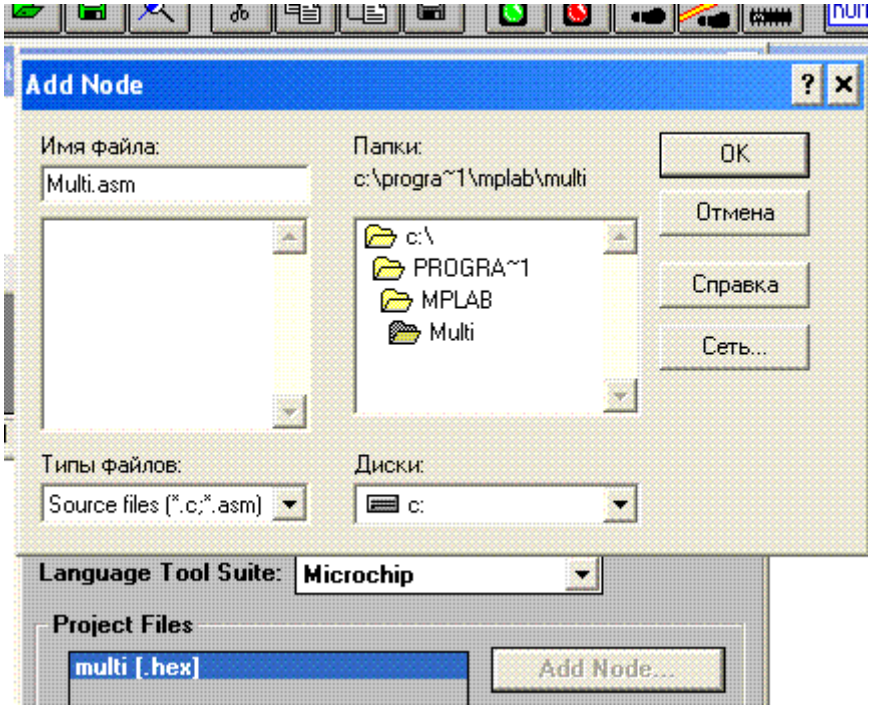
Щелкаем по ***OK***.
Файл проекта с расширением **.PJT** создан и "ушел" в папку **Multi**.
Одновременно открывается окно ***Edit Project***, в котором предлагается создать "пустышку" **HEX-файла**.
Но перед этим, на всякий случай, нужно убедиться, что в окошке ***Divelopment Mode*** установлено: **MPLAB SIM PIC16F84A** (то, чем мы занимались в настройках), а в окошке ***Language Tool Suite*** установлено: **Microchip**.
Создаем "пустышку".
В списке ***Project Files*** щелкаем по строке **multi(.hex).**
После этого, строка выделяется и кнопка ***Node Properties...*** активизируется.



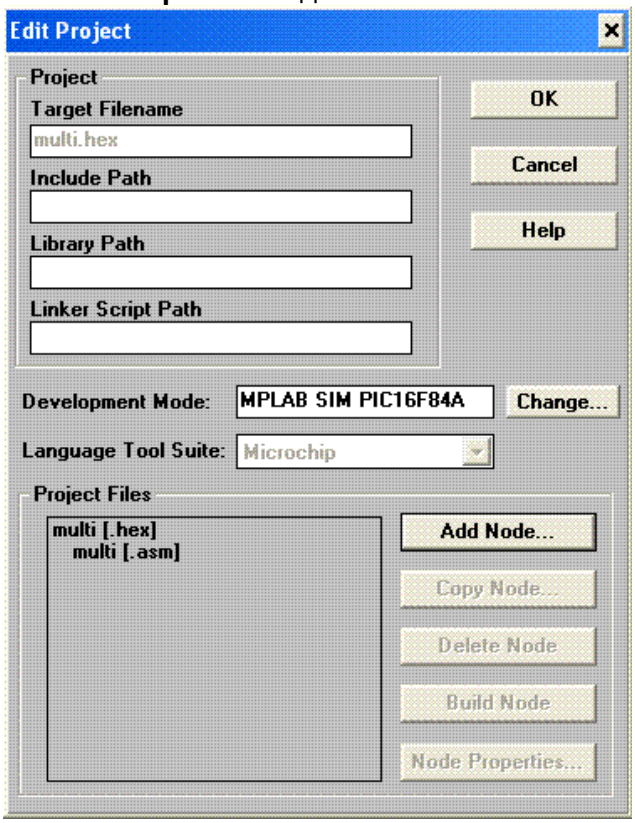
Щелкаем по ней.
Открывается окно ***Node Properties***.



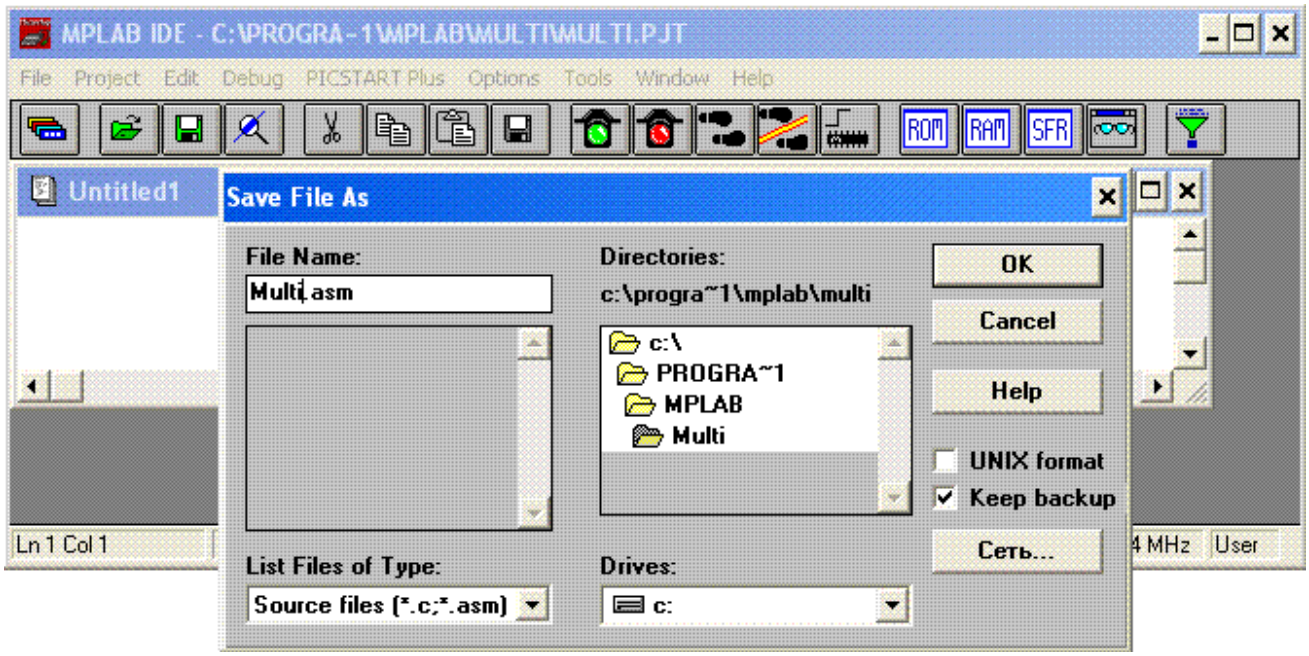
Менять здесь ничего не нужно (настройки по умолчанию).
Единственное, что нужно сделать – убедиться в том, что в окошке ***Node*** установлено: **MULTI.HEX**, а в окошке ***Language Tool*** установлено: **MPASM**.
Щелкаем по ***OK***. "Пустышка" HEX - файла создана и "ушла" в папку **Multi**. Опять появится окно ***Edit Project***. Теперь создаем "пустышку" **ASM-файла**.
Щелкаем по кнопке ***Add Node...*** Раскрывается окно ***Add Node***.
Это то же самое , что и окно ***New Project***, только предназначенное для создания **ASM-файла**, поэтому нужно сделать в точности то же самое, что и в окне ***New Project***.
Единственное различие может быть в том, что, если Вы указываете название файла с расширением, то естественно, что оно должно быть не **.PJT**, а **.ASM**
Чтобы не ошибиться, настучите **Multi.asm** и дело с концом.



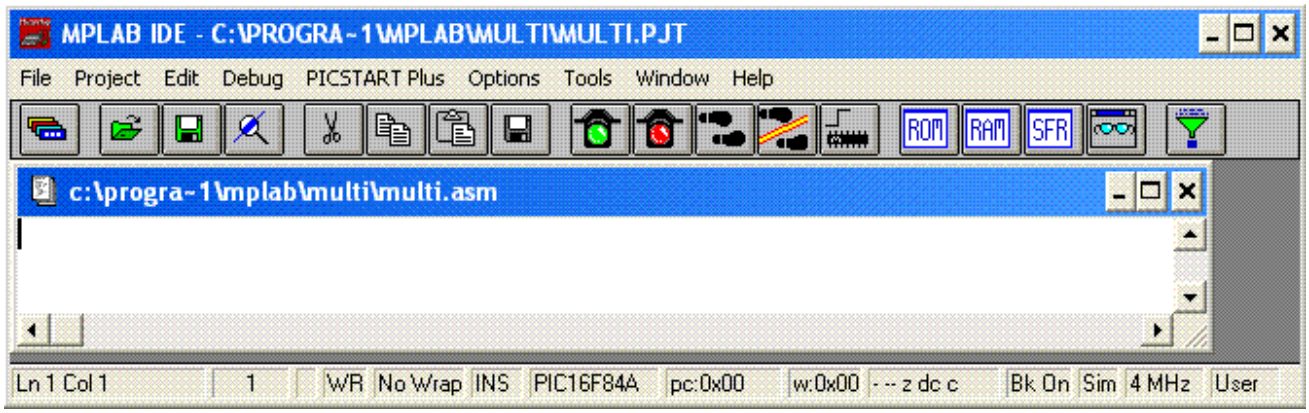
Щелкаем по ***OK***. "Пустышка" **ASM-файла** создана. Опять появится окно ***Edit Project***.



В списке этого окна Вы увидите, что к **HEX-файлу** добавился **ASM-файл**.
Щелкните по ***OK***.
Окно ***Edit Project*** закроется, и Вы останетесь "один на один" с пустым файлом **Untitled...** .
Необходимо сделать его пригодным для работы по составлению текста программы (следовательно, речь идет о файле с расширением **.ASM**).
Для этого необходимо назвать его **Multi.asm** и "вписать" его в ранее созданную "пустышку" **ASM-файла** (см. выше).
Делается это так: щелкаем по слову ***File*** в главном меню, а затем - по строке выпадающего списка ***Save As..*** .
Открывается уже знакомое нам окно, только с названием ***Save File As***.
Выбираем папку **Multi** и определяем название файла (**Multi.asm**).



Здесь Вы еще увидите нечто, перед которым можно поставить или не поставить галочку.
Ничего менять не нужно.
Щелкаем по ***OK*** и опять остаемся наедине с пустым файлом, только теперь он "прописан" в папке **Multi** под названием **multi.asm** (см. адресную строку окна файла).



Обратите внимание: и в списке окна ***Edit Project***, и в адресной строке созданного нами **ASM - файла**, большая буква **M**, в названии файла **Multi.asm**, "превратилась" в маленькую **m**.
Пусть Вас это не смущает, так как "что в лоб, что по лбу - все едино".
Особо дотошные могут работать только в нижнем регистре, а я, в этом смысле, привык работать по старинке. Каюсь, грешен, но ничего с собой поделать не могу.
Итак, проект **Multi** создан, и перед Вашими глазами сейчас находится "поле предстоящих сражений" (файл **Multi.asm**).
Именно на этом, пока пустом, "белом листе" мы и будем заниматься тем, что называется

программированием.
"Для полного счастья", нужно потренироваться в закрытии и открытии проекта.
Сейчас проект открыт.
Для того чтобы закрыть проект, в главном меню, щелкните по слову ***Project,*** и в выпадающем списке, щелкните по строке ***Close Project*** (закрыть проект).
При первом закрытии, программа может ничего не спросить, но при последующих закрытиях проекта, она может спросить: сохранить или нет внесенные изменения?
Тут, как говорится, дело хозяйское.
В этом случае нужно щелкнуть или по ***Yes*** или по ***No*** и проект закроется.
Для того чтобы открыть проект, в главном меню, щелкните по слову ***Project,*** и в выпадающем списке щелкните по строке ***Open Project*** (открыть проект).
Откроется знакомое Вам окно, только с названием ***Open Project***, в котором нужно указать папку, в которой находится искомый проект (в нашем случае - папка **Multi**).
После открытия этой папки, в списке (под окошком File Name) появится название файла проекта.
Если это то, что нужно, то необходимо щелкнуть по строке с названием файла проекта, а затем - по ***OK***.
Окно ***Open Project*** закроется, и в нашем случае, появится предупреждение, что **HEX-файл** "пустой", то есть, текст программы не ассемблировался, что и соответствует действительности.
Говорим спасибо программе за "бдительность" и щелкаем по ***OK***.
После этого, проект откроется, и Вы увидите, в нашем случае, пустой файл **Multi.asm** во всей своей "боевой готовности".
Потренируйтесь. Создайте несколько своих пустых проектов в папке **Pic**.
Примечание: если производилось ассемблирование (**HEX-файл** не пустой), то указанное выше предупреждение, не выдается, и **ASM-файл** открывается сразу.
Если после работы с текстом программы, производилось ассемблирование, и сразу же после этого проект закрывается, то он закроется сразу после ответа на вопрос программы: "Сохранить изменения или нет"?
Если после последнего, перед закрытием проекта, ассемблирования, в текст программы были внесены изменения, то **MPLAB** напомнит Вам о необходимости ассемблирования перед закрытием проекта, проведет его (щелкнуть по ***Yes***), опять задаст вопрос: **"Сохранить изменения или нет"?**, и после ответа на него, закроет проект.
Прочитал написанное и сам удивился: не думал, что объяснение такой относительно несложной процедуры потребует столько много слов.
Но проще как-то не получается.
Надеюсь, что не смотря на это, объяснение, все-таки, получилось доходчивым и понятным.
После того, как Вы "набьете руку", открытие нового проекта будет занимать у Вас не более 1 - 1,5 минут, и Вы сами убедитесь, как это относительно просто.

"Загрузка" текстов программ в текстовый редактор.

Речь пойдет **о "загрузке" файлов с расширением .ASM** (тексты программ).
Если необходимо просто просмотреть текст той или иной программы (например, Вы скачали в Интернете **ASM-файл** какой-нибудь программы и хотите с ней разобраться), то проекта создавать не нужно.
Если нужно изменить текст этой программы и/или создать ее **HEX-файл**, то необходимо "загрузить" (скопировать) текст этой программы в текстовый редактор (в "пустое" окно нового проекта).
Могут быть и другие необходимости.
Давайте разберемся с "технологией этого действа".
Для работы, необходим какой-нибудь **ASM-файл**.
Я буду использовать файл программы **Multi.asm**
Примечание: ранее Вы познакомились с текстом этой программы в формате .DOC (см. выше).
Это сделано для удобства тех читателей, которые не установили на своем компьютере **MPLAB**.

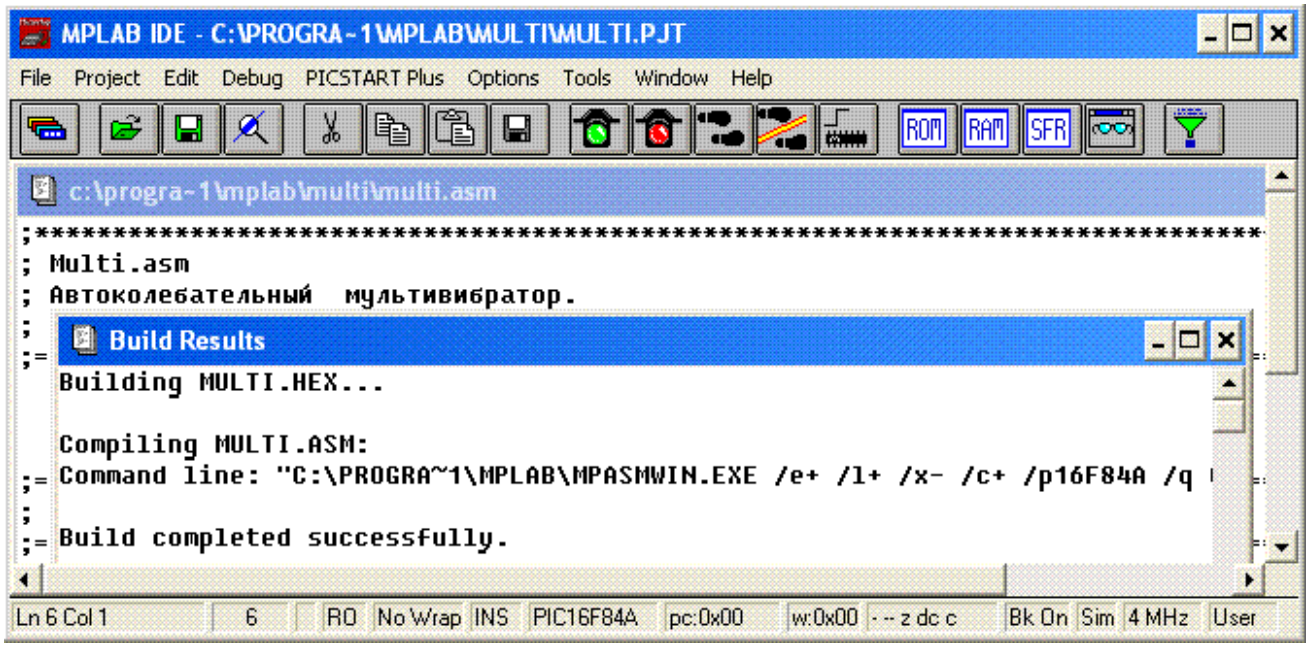
Файл программы Multi.asm прилагается (находится в папке ***"Тексты программ"*****).**

С ним и будем работать.
Далее, все файлы текстов программ и подпрограмм, публикуемые в "Самоучителе...", будут иметь расширение **.ASM**.
Итак, открываем файл **Multi.asm** вне проекта.
Запустите **MPLAB**.
Если программа предложит Вам открыть какой-то более ранний проект **MPLAB**, то откажитесь, щелкнув по ***No***.
Перед Вами → пустое окно **MPLAB**.
В главном меню программы, щелкните по слову ***File*** и далее, по строке ***Open*** (в выпадающем списке).
Откроется знакомое Вам окно MPLABовского проводника, только с названием ***Open Existing File***.
Найдите то место, где у Вас лежит файл **Multi.asm**
После этого, Вы увидите его название в списке проводника.
Щелкните по строке с этим названием, а затем по ***OK*** (или двойной щелчок по строке).
После этого, Вы увидите окно с текстом выбранной Вами программы.
Так как предполагается работа, связанная с изменением текста этой программы, и мы ранее (см. выше) создали "пустой" проект, то необходимо **загрузить текст программы в проект**.
Делается это так:
Не закрывая окно с текстом программы, нужно открыть проект **Multi**.
Вы раньше это уже делали, так что сделайте это сейчас самостоятельно.
В случаях появления сообщений об ошибках ***Project Error*** и ***Import Error*** (**ASM-файл** проекта пустой и ранее ассемблирования не производилось), необходимо их убрать, щелкнув по ***OK***.
После этого, откроется второе окно "заготовки" **ASM-файла** проекта.
Оно пустое.
Нужно скопировать в него содержимое окна, которое находится за ним (перенести содержимое текста программы **Multi.asm** в "пустой" проект).
Переключения между окнами и сам процесс копирования ничем не отличается от того же самого, при работе в "Виндах".
Копирование производится через буфер обмена.
А именно: активизируете окно с текстом программы.
В главном меню **MPLAB,** щёлкаете по слову ***Edit*** и в выпадающем списке, щёлкаете по ***Select All*** (выделить всё).
Далее опять щёлкаете по ***Edit***, а в выпадающем списке, по ***Copy*** (текст программы скопировался в буфер обмена).
Затем активизируете "пустое" окно.
Далее, опять щелкаете по ***Edit***, а в выпадающем списке, по ***Paste*** (вставить).
Всё. Копирование текста программы **Multi.asm** в проект произведено, и Вы видите этот текст в бывшей "пустышке" **ASM-файла** проекта.
Таким образом, "пустышка" **ASM-файла** "превратилась в полноценный" **ASM-файл** проекта, с которым можно работать.
После этого, окно с текстом программы, из которого производилось копирование, становится ненужным и его можно убрать обычным для "Виндов" способом, то есть, щёлкнув по кнопке ***Закрыть*** в правом верхнем углу этого окна.
Далее, для удобства восприятия, разверните оставшееся окно, щёлкнув по кнопке ***Развернуть*** (находится в правом верхнем углу этого окна).
После этого, Вы увидите текст программы **Multi.asm** в полной "боевой готовности".
Так как этот текст в ПИК не зашьёшь, то начинаем разбираться с HEX - файлом ("прошивкой").
В проекте "лежит пустышка" ("заготовка") **HEX-файла**.
Для того чтобы она "превратилась в полноценный" **HEX-файл** (то есть, наполнилась содержанием), необходимо ***проассемблировать текст программы***.
Текст программы в наличии, так что никаких преград, для выполнения этого действия, нет.

Создание HEX-файла.

Ассемблирование (компиляция) выполняется очень просто:
В главном меню **MPLAB** щелкаете по слову ***Project***, а в выпадающем списке, по строке

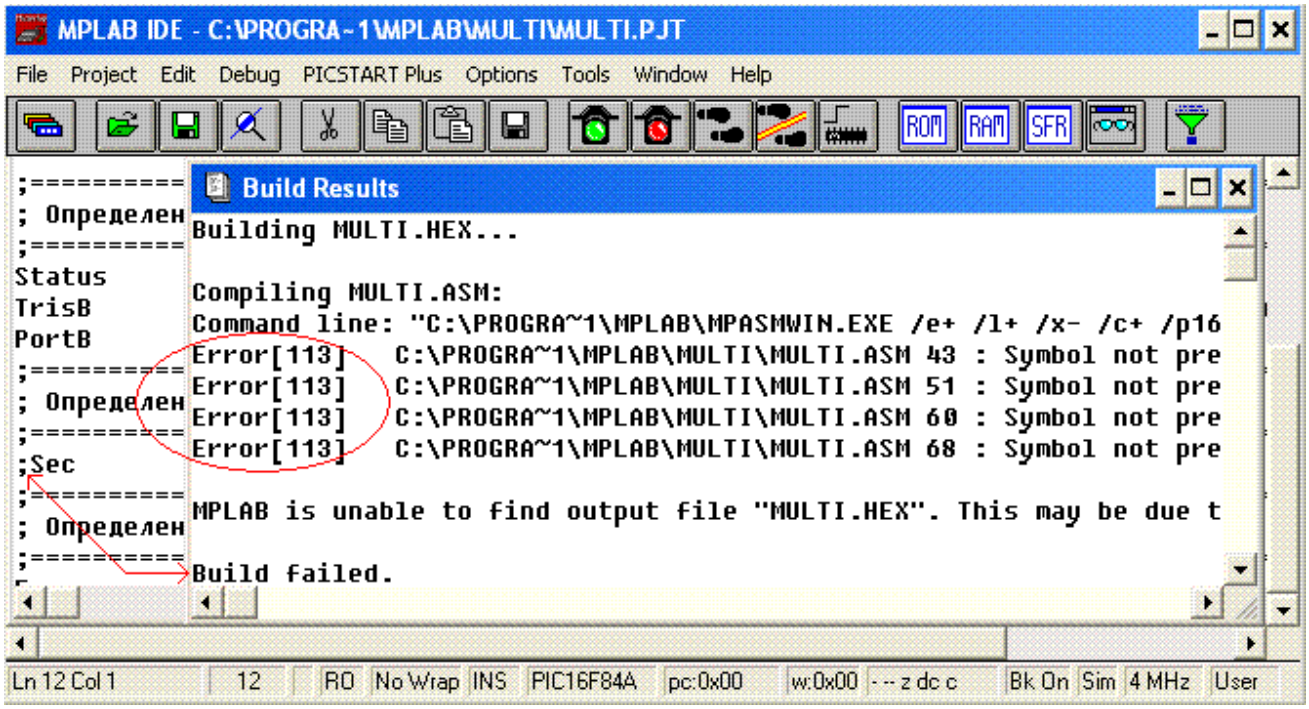
Build Node.
Далее, ни во что не вмешивайтесь, а только ждите окончания процесса.
MPLAB "все сделает в наилучшем виде".
В конечном итоге, Вы увидите окно ***Build Results*** с текстом, последняя строка которого должна выглядеть так: **`Build completed successfully`**.



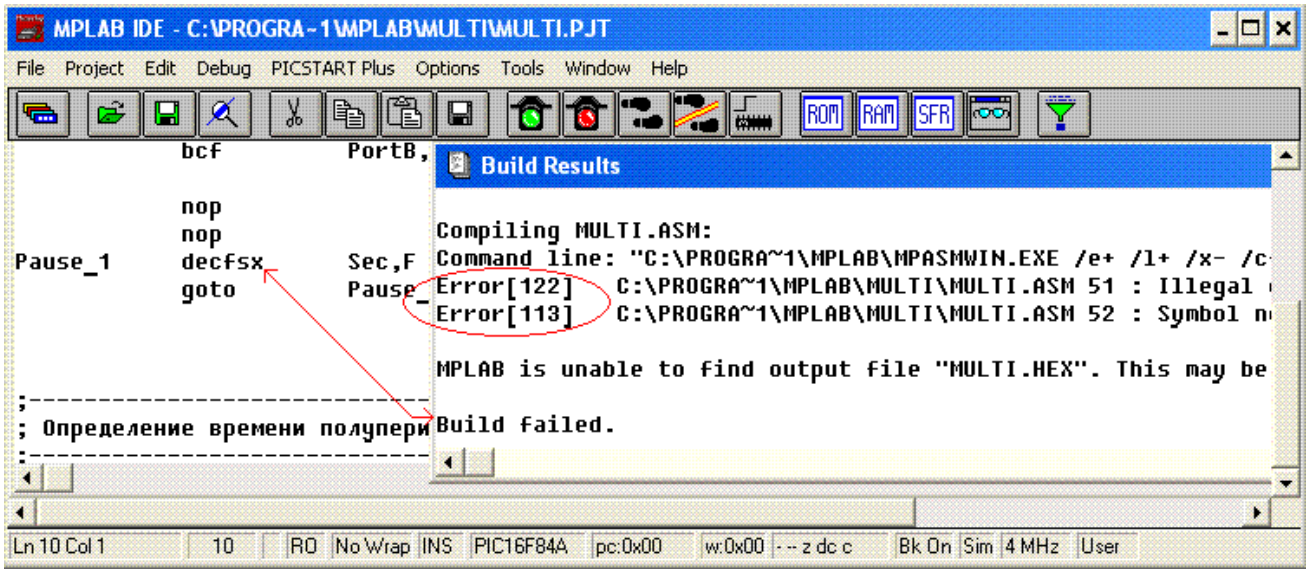
Это соответствует случаю **безошибочного** составления текста программы.
После этого окно ***Build Results***, со спокойной совестью, можно закрыть.
Если в тексте программы будут ошибки, то в окне ***Build Results*** будет выведен их список, а содержание последней строки будет иным.
Разбор этого варианта будет ниже.
Итак, сообщение об отсутствии ошибок получено, что говорит о том, что пустышка **HEX-файла** проекта "превратилась" в полноценный **HEX-файл** (**Multi.hex**), который можно "загружать" в программу, обслуживающую программатор, а значит и "зашить" в ПИК.
Желающие (имеющие программатор) могут это сделать.
Получится автоколебательный мультивибратор со стабильной частотой 5кГц.
Итак, Вы приобрели для себя новую возможность.
Теперь, имея файл с расширением **.ASM** (текст программы), **Вы можете самостоятельно создать файл "прошивки"** (**HEX-файл**), а это уже не мало.
Даже не имея "штатного" **ASM-файла**, а имея "нештатный" текст программы (например, опубликованный в какой-нибудь книге), Вы можете его "настучать" в текстовом редакторе **MPLAB** (с соблюдением синтаксических правил написания текста программы) и после этого, создать **HEX-файл** этой программы.
Для этого даже не нужно быть программистом в буквальном смысле этого слова.
Программист же, должен, описанные Выше операции, производить "в автомате" (на уровне условных рефлексов).
Поэтому, тренируйтесь.
При составлении программ, мозги не должны отвлекаться на такие "технологические вещи".
Для них и так, кроме этого, будет достаточно другой, более значимой, работы.

Работа с текстом программы при наличии ошибок.

Посмотрим, что произойдет, если в тексте программы допущена ошибка.
Искусственно введу ее.
Например, в "шапке" программы **Multi.asm**, поставлю точку с запятой перед названием регистра **Sec** (напоминаю, что **MPLAB** "ничего не видит" после точки с запятой).
Ассемблируем текст программы и получаем сообщение о 4-х ошибках, с нижней строкой в конце, в которой сообщается о "неудачном" ассемблировании (**MPLAB** "ушел в отказ").

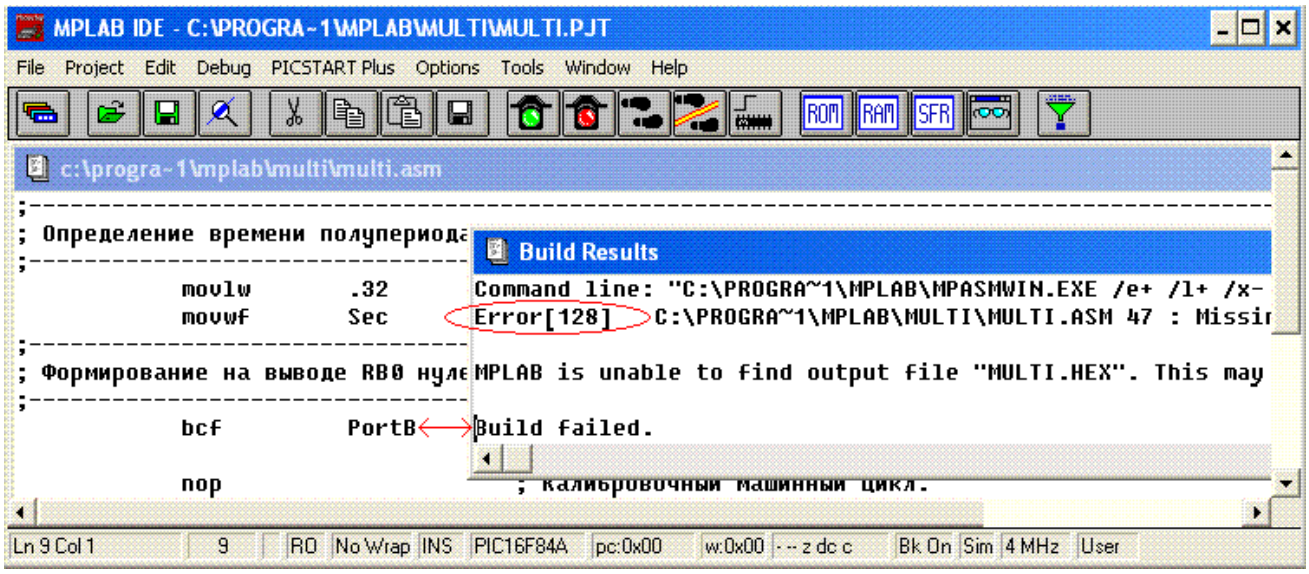


Если Вы сделаете двойной щелчок по строке с ошибкой, то курсор покажет Вам эту строку в тексте программы.
Так как ошибка заключается в том, что после блокировки, регистр **Sec** перестал быть "прописанным" в "шапке" программы, то ошибочными будут считаться все строки программы, в которых упоминается регистр **Sec**.
Это и есть объяснение того, что в данном случае, количество ошибок равно четырем.
Уберите точку с запятой перед регистром **Sec** и после этого, еще раз проассемблируйте текст программы.
Вы получите сообщение о безошибочном ассемблировании.
Еще один случай.
Предположим, что Вы ошиблись при наборе текста программы и вместо команды **decfsz** (в подпрограмме **Pause_1**) "настучали" **decfsx** (нажали, соседнюю с **z**, кнопку клавиатуры).
Введите эту ошибку и затем проассемблируйте текст программы.



Вы получите сообщение о 2-х ошибках: грамматическая ошибка в слове команды и ошибка в подпрограмме **Pause_1** (в которой находится ошибочная команда).
Пощелкайте по сообщениям об ошибках, и Вы увидите, что в первом случае, **MPLAB** покажет на строку с ошибочной командой, а во втором случае, на строку, в которой находится название подпрограммы.

Исправьте эту ошибку и проассемблируйте текст программы.
Вы получите сообщение о безошибочном ассемблировании.
Третий случай.
Например, Вы забыли указать номер бита, к которому обращается бит-ориентированная команда **BCF** при формировании нулевого уровня на выводе **RB0**, и вместо **bcf PortB,0** "настучали" **bcf PortB**.
Введите эту ошибку в текст программы и проассемблируйте его.

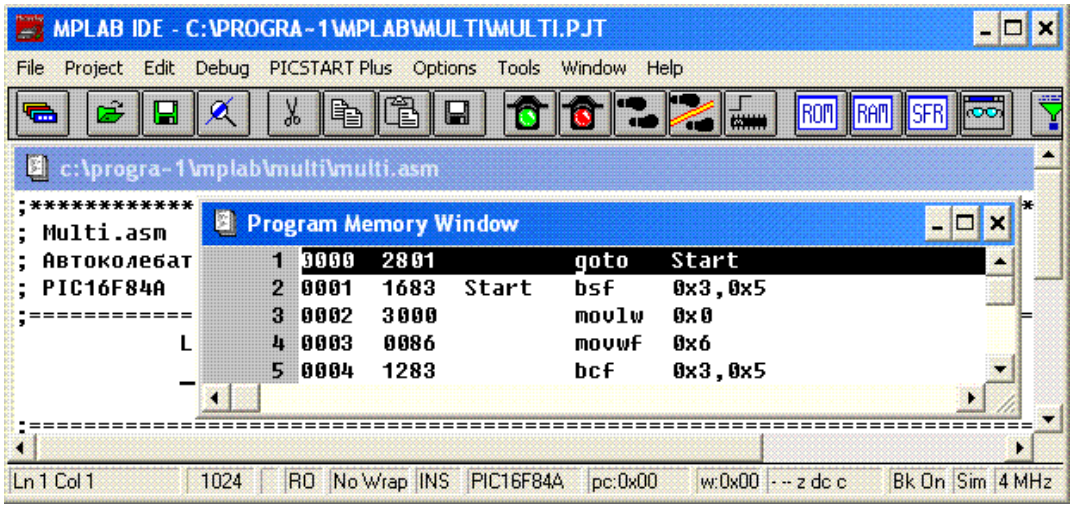


Вы получите сообщение об ошибке типа: в функции отсутствует аргумент (то есть, чего-то не хватает).
Устраните ошибку и проассемблируйте текст программы.
Вы получите сообщение о безошибочном ассемблировании.
При проведении такого рода "экспериментов", обратите внимание на изменение цвета полосы загрузочного индикатора, в конце процесса ассемблирования, с **зеленого** (все в норме) на **красный** (в случаях наличия хотя бы одной ошибки).
В случаях отсутствия ошибок, цвет полосы не меняется и после завершения процесса ассемблирования, остается **зеленым**.
Можно привести еще множество подобного рода примеров.
Если есть желание, то самостоятельно введите другие ошибки и посмотрите, какие сообщения Вам выдаст **MPLAB**.
Не забывайте только потом исправлять их.
Смысл сказанного выше: если в тексте программы допущена ошибка, то после завершения процесса ассемблирования, MPLAB укажет на нее и подскажет тип ошибки. Устранением ошибок MPLAB не занимается, это личное дело программиста
Обращаю Ваше внимание на следующее: **MPLAB** не всесилен.
Он не может обнаружить ошибки, связанные с неправильным замыслом программы и стратегией ее построения (функциональные ошибки). Это дело программиста.
MPLAB контролирует только "технологию" текста программы: наличие или отсутствие грамматических ошибок, соблюдение или нет синтаксических правил написания программы, правил переходов, указания аргументов и т. д.
Сообщение об отсутствии ошибок означает только соблюдение этих "технологических" правил и не является гарантией реализации замыслов программиста.
По той простой причине, что "технология" может быть соблюдена, но при этом, замысел программы может быть "кривым" (наличие функциональных ошибок).
Пример функциональной ошибки: "несанкционированный уход" рабочей точки программы в "вечное кольцо", без возможности из него выйти ("зависание"/"глюк").
Таким образом, для того чтобы создать "полноценную", рабочую программу, программист должен реализовать "жизнеспособный", стратегическо-тактический замысел программы, при условии соблюдения всех "технологических" правил ее составления.
Пока мы занимаемся только "технологией", так как без нее невозможна реализация любого

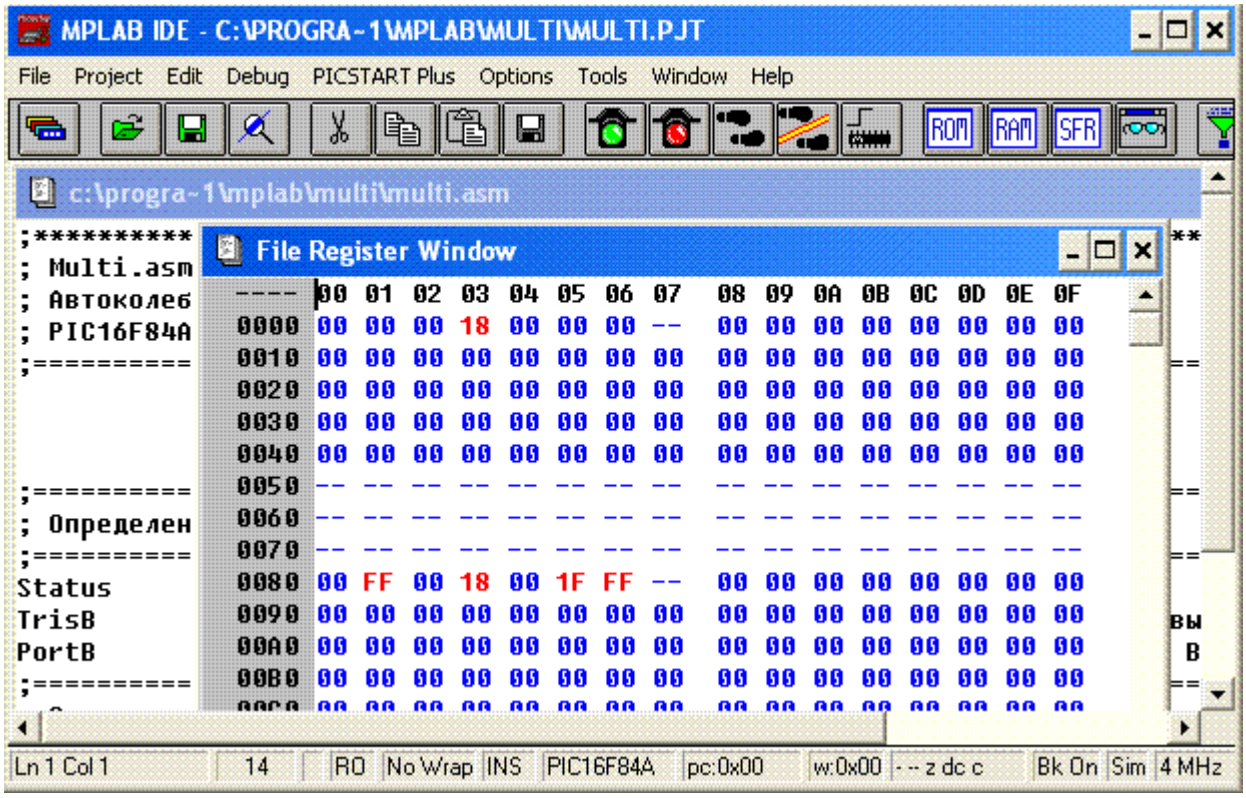
замысла.
Если кто-то из Вас "рвется в бой" (в смысле желания побыстрее перейти к творчеству), то советую "обуздать свои порывы" и заняться, пусть и не самой интересной, но абсолютно необходимой, рутинной работой, а иначе Вам гарантирован повышенный процент неудач.

Разбираемся с симулятором

Прежде всего, симулятор позволяет заглянуть в "начинку" ПИКа и увидеть, что там происходит при выполнении команд программы.
При этом следует четко осознавать, что программист будет видеть "начинку" не реального ПИКа, а того "виртуального" (как бы, "идеального") ПИКа, который создается в **MPLAB** (в ее симуляторе).
Если выбран **PIC16F84A**, то Вашему вниманию и будет предоставлена "начинка" "виртуального" **PIC16F84A**, работающая по всем законам реального **PIC16F84A**.
Если реальный ПИК нужно "прошить" в программаторе, то "виртуальный" ПИК "прошивается" автоматически. При ассемблировании.
И какого-то внешнего устройства, типа программатора, для этого не нужно.
В симуляторе, имитируется вся "начинка" ПИКа, с наглядным показом того, что в ней происходит по ходу отработки программы.
В нем можно сымитировать и внешнее воздействие.
Это позволяет обнаруживать и устранять ошибки (в том числе и функциональные), а также производить отладку как составных частей программы, так и всей программы.
Короче, "палочка - выручалочка".
Таким образом, сначала решаются "технологические" задачи, а когда они решены (сообщение о безошибочном ассемблировании), программист, с помощью симулятора, может проверить соответствие замысла программы тому, что желаемо.
Если с этим все в порядке, то можно произвести отладку ее характеристик (например, временных. Если это необходимо).
В подавляющем большинстве случаев, программа, прошедшая через "горнило" симулятора ("прогнанная" через него во всех возможных режимах), будет нормально работать и в реальном ПИКе.
Давайте заглянем в "начинку" "виртуального" **PIC16F84A**.
Программа **Multi.asm** проассемблирована. Это означает то, что в "виртуальный" ПИК уже "зашита" программа **Multi** и с ней можно работать в симуляторе.
Обратите внимание на группу из 4-х кнопок в правой части строки с пиктограммами.
Это кнопки с надписями ***ROM, RAM, SFR*** и ***кнопка с нарисованными на ней очками***.
Если проект **Multi** у Вас не открыт, то откройте его.
Ассемблирования производить не нужно (если последнее ассемблирование было безошибочным и после него в текст программы не вносились изменения. Если это не так, то проассемблируйте текст программы, можно даже по принципу: "на всякий случай").
Щелкните по кнопке ***ROM***. Откроется окно ***Program Memory Window*** (содержимое памяти программ).



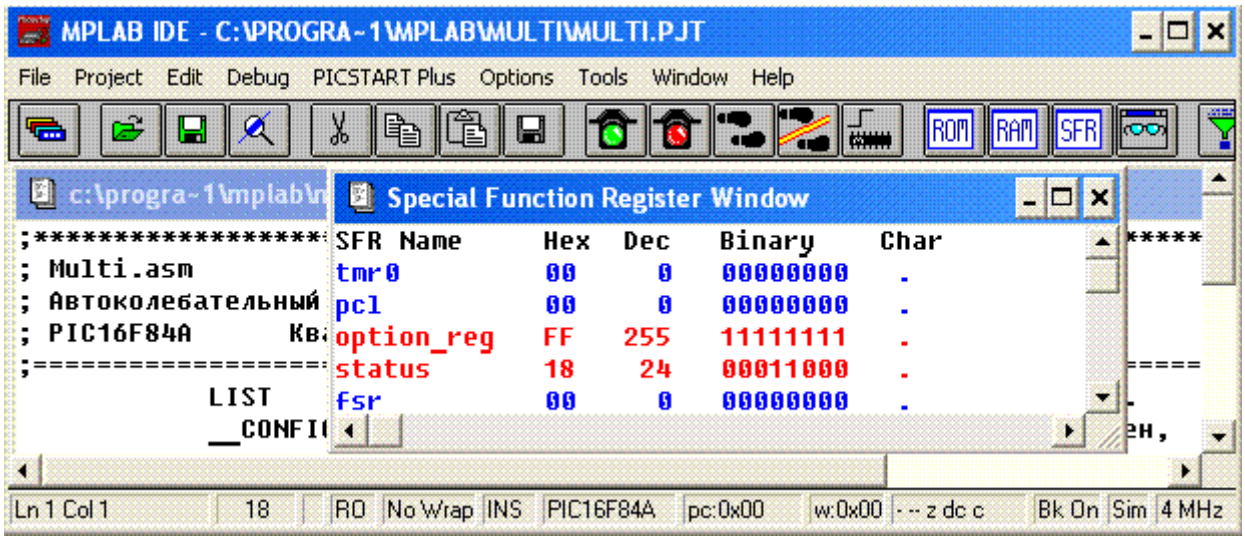
В нем:
1-й столбец: номера ячеек памяти программ, в 10-чной системе исчисления.
2-й столбец: адреса ячеек памяти программ, в 16-ричной системе исчисления.
3-й столбец: пока (да и потом тоже) нам не нужен. Не обращайте на него внимания.
4-й, 5-й и 6-й столбцы: собственно говоря, сама программа в "чистом" виде (полное разложение на команды), то есть, без всего того, что расположено правее точек с запятой.
То, что Вы видите, есть содержимое памяти программ, с указанием адресов и порядковых номеров команд рабочей части программы.
Напоминаю, что в части касающейся директив, исполняемых в рабочей части программы, в память программ "закладываются" не они, а их разложения на команды.
Хотя, в рабочей части программы **Multi.asm**, директивы и не используются, но напомнить об этом не лишне.
Пояснение: в память программ ПИКов "закладываются" не команды на языке ассемблер, а машинные коды этих команд. И слава Богу, что **MPLAB** "ограждает нас от этой напасти"...
После "прошивки" реального **PIC16F84A,** в его памяти программ, именно по этим адресам, и в такой же последовательности (по направлению сверху вниз), будут располагаться машинные коды команд программы **Multi.asm**.
В окне ***ROM***, можно определить, какое именно количество ячеек памяти программ занимает программа (в данном случае, 20) и каков объем памяти программ используемого ПИКа (в данном случае, 1024 ячейки).
Закройте окно ***Program Memory Window***.
Щелкните по кнопке ***RAM***.
Откроется окно ***File Register Window***.



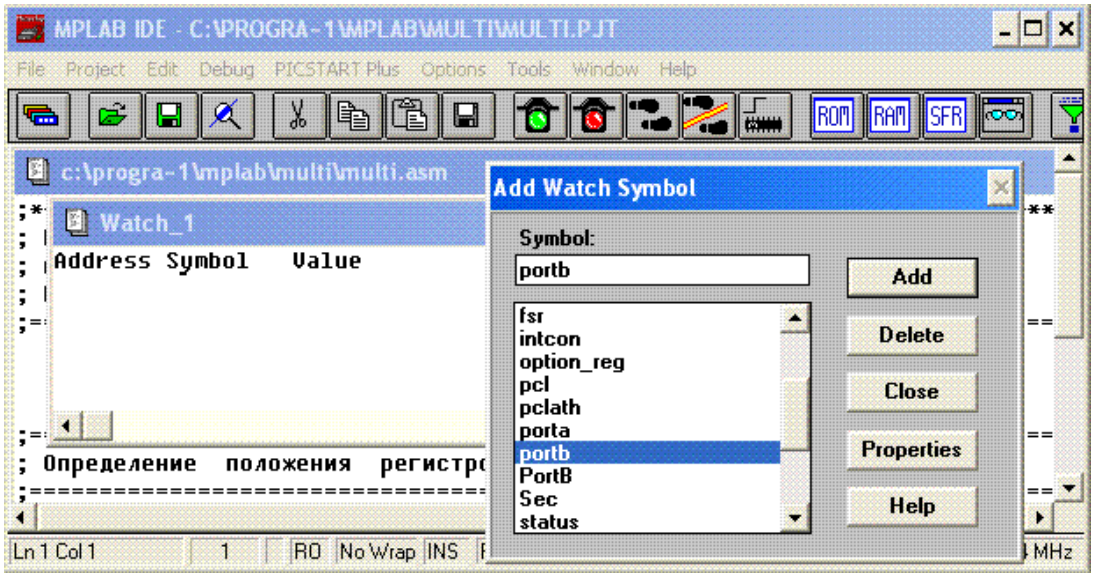
В нем Вы видите область оперативной памяти. По образу и подобию, имеющейся у Вас, распечатки области оперативной памяти.
Сравните эту распечатку (с учетом того, что в распечатке указаны не все регистры общего назначения) с содержимым окна ***File Register Window***.
В "глобальном смысле", Вы увидите примерно одинаковую картину, только с различными, взаимно дополняющими друг друга, "формами наполнения".
Используя окно ***File Register Window*** и распечатку области оперативной памяти (в комплексе), Вы можете свободно ориентироваться в области оперативной памяти.
Например, сверяясь с распечаткой области оперативной памяти, в окне ***File Register Window***, Вы сможете увидеть "числовую начинку" любого из регистров области оперативной

памяти. Причем, по состоянию на момент завершения исполнения любой команды программы (!!!).
Напоминаю, что все регистры, находящиеся в области оперативной памяти, **8-битные**, и поэтому они работают в числовом диапазоне **от 00h до FFh** (от .0 до .255).
В окне ***File Register Window***, содержимое всех регистров отображается **только в 16-ричной форме исчисления**.
Таким образом, в окне File Register Window, можно отслеживать изменения содержимого всех регистров области оперативной памяти.
Обратите внимание на следующую особенность: если после исполнения той или иной команды, произошло изменение содержимого того или иного регистра, то до момента исполнения следующей команды программы, это содержимое будет выделено **красным цветом.**
Если таких изменений не происходит, то оно (содержимое) будет выделено **синим цветом**.
Таким образом, при пошаговом исполнении программы (об этом, позже), четко видно, содержимое какого именно регистра изменилось и как именно ("в числовом эквиваленте") оно изменилось.
Пример: в регистр **Sec**, расположенный в области оперативной памяти, например, по адресу **0Ch**, командой **movwf Sec**, из регистра **W**, копируется заранее заложенная в него (в **W**) константа, например **.100**.
Пусть до момента исполнения команды **movwf Sec**, в регистре **Sec** "лежало", например, число **0** (напоминаю, что любое число, до 9-ти включительно, можно указывать в команде без атрибутов систем исчисления).
При этом, в окне ***File Register Window,*** Вы увидите:
- До выполнения команды **movwf Sec**, в ячейке области оперативной памяти с адресом **0Ch,** будет "лежать" число **00,** выделенное **синим цветом**.
- В момент исполнения этой команды, число **00** заменится на число **64**, которое будет выделено **красным цветом** (произошло изменение).
- Если, после этого, следующая команда программы не производит изменения содержимого регистра **Sec**, то после выполнения этой команды, число **64** изменит свой цвет с **красного** на **синий**.

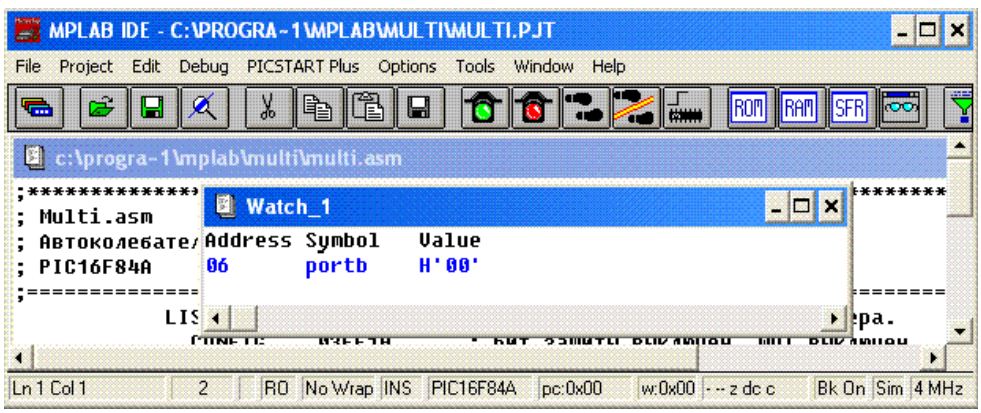
Почему 64, а не 100?
Вот Вам и типичный пример перевода чисел из одной системы исчисления в другую (из 10-чной в 16-ричную и наоборот).
Откройте, имеющуюся у Вас, таблицу перевода чисел из одной системы исчисления в другую и убедитесь, что числу **.100** (10-чная система исчисления) соответствует число **64h** (16-ричная система исчисления).
Если Вы хотите устранить этот "разнобой", то в командах программы, применяйте числа в 16-ричной системе исчисления. Это кому как удобнее.
Закройте окно ***File Register Window***.
Щелкните по кнопке ***SFR***. Откроется окно ***Special Function Register Window***.



Если в предыдущем окне Вы видели содержимое всех регистров области оперативной памяти, в 16-ричной системе исчисления, то в этом окне Вы видите содержимое только регистров специального назначения, в трех системах исчисления.
Окно ***Special Function Regisrer Window*** можно использовать в качестве конвертора систем исчисления.
В приложении к числам, зафиксировавшимся в регистрах специального назначения, на том или ином этапе пошагового исполнения программы. Причем, для всех сразу.
Для начинающих, это окно может быть "палочкой - выручалочкой".
Программисты, которые "набили себе руку", этим окном пользуются не часто.
В основном, они работают в окне ***File Register Window***.
Выделение цветом, описанное выше, работает также и в окне ***Special Function Register Window.***
Закройте окно ***Special Function Register Window***.
Щелкните по кнопке с ***нарисованными на ней очками***. Откроется окно ***Add Watch Symbol***.
Если программист хочет выборочно видеть содержимое только тех регистров, которые его интересуют, то это как раз тот случай.
В окне ***Add Watch Symbol,*** Вы увидите список регистров специального назначения и "прописанных" регистров общего назначения, плюс регистр **W**.
Названия регистров расположены в алфавитном порядке.
Например, нас интересует содержимое регистра **PortB**.
Для этого, в списке, нужно найти название регистра **PortB** и щелкнуть по строке с этим названием.
После этого кнопка ***Add*** (добавить) активизируется.



Затем, нужно щелкнуть по этой кнопке, а затем - по кнопке ***Close***.
Окно ***Add Watch Symbol*** закроется, и Вы увидите, что в окне ***Watch*** (перед этим, оно было на заднем плане) "появятся начинка" регистра **PortB**.

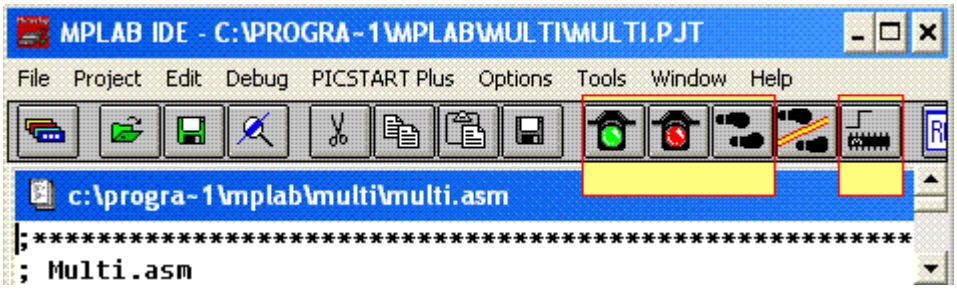


Таким образом, по желанию программиста, в окне ***Watch,*** можно сформировать список из любого количества задействованных в программе регистров (с целью дальнейшего контроля за их содержимым).
Обратите внимание на то, что содержимое регистра **W** можно увидеть только в окне ***Watch*** (естественно, после его выбора и добавления в рабочий список).
В окне ***Watch***, содержимое регистров отображается только в 16-ричной форме исчисления.
Выделение цветом (см. выше) также работает.
Лично я, жму на кнопку с очками только в том случае, если мне нужно увидеть содержимое регистра **W**.
Содержимое остальных регистров можно увидеть в окне ***File Register Window***.
Вы можете работать по-другому. Это личный выбор программиста. Зависит от привычки.
При закрытии окна ***Watch***, "программа спросит", нужно или не нужно сохранить файл с данными этого окна?
Жмите на ***No***.
Проще еще раз выбрать то, что нужно, чем "возиться" с открытием этого файла.

Режимы работы симулятора. Точки остановок.

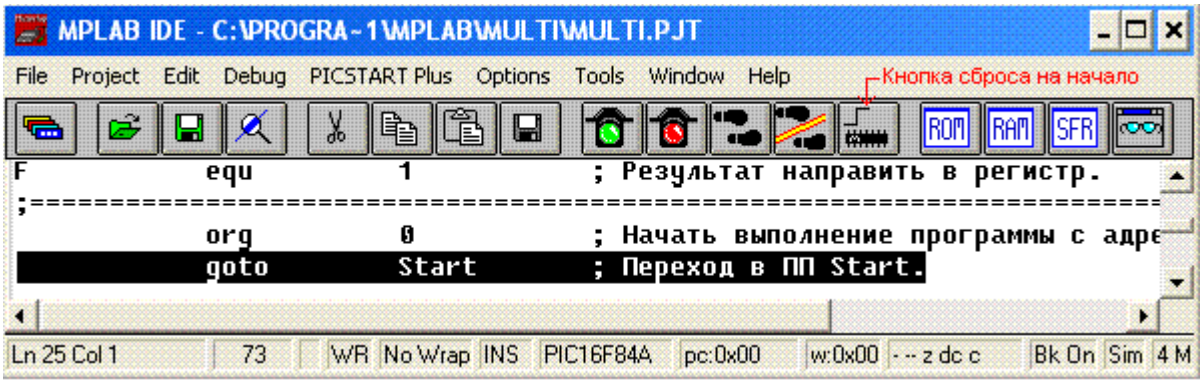
В симуляторе, исполнение программы осуществляется в одном из двух основных режимов:
- **в режиме пошагового исполнения программы,**
- **режиме автоматического исполнения программы.**

Органы управления:
Откройте проект **Multi** и обратите внимание на отдельную группу из 5-ти пиктограмм, с **красным** и **зеленым** светофорами.
Четвертую слева пиктограмму (следы с **желто-оранжевой** полосой) можно проигнорировать (без нее вполне можно обойтись).
Остаются 4 кнопки (пиктограммы).



В большинстве случаев, отладка программы начинается с команды **goto Start** (нулевой адрес в памяти программ), и она происходит последовательно.
Это означает то, что в большинстве случаев (но не во всех), нежелательно начинать отладку (исполнение программы), например, с середины (условно) программы.
Почему?
Потому, что не будет отработана та часть программы, которая должна быть отработана "до того" (в ней могут формироваться данные, влияющие на ход дальнейшего исполнения программы), и по этой причине, можно банально "проколоться".
Чтобы не случилось этого "конфуза", до этой "середины" необходимо сначала "дойти", то есть, полноценно исполнить все предшествующие команды программы.
В частности, для этого и нужны режимы пошагового и автоматического исполнения программы.
Эти рассуждения относятся к случаям, когда нужно произвести отладку подпрограммы (или группы подпрограмм), находящейся внутри программы и "привязанной" к результату работы предшествующей части программы.
Такая "привязка" может быть, а может и не быть.
Если такая "привязка" есть, то см. выше, а если ее нет, то можно сразу, без отработки предшествующей части программы, "прыгать" на начало исполнения той подпрограммы, работу которой нужно отследить.
При этом применяется то, что я называю "уловками", и о которых я расскажу позднее.
Для установки рабочей точки программы на начало программы (на команду **goto Start**),

используется крайняя правая кнопка указанной выше группы пиктограмм.
Я, честно говоря, должным образом не оценил "глубинного смысла" того, что нарисовано на этой кнопке, и поэтому я буду ее называть ***кнопкой сброса программы на начало***.
Щелкните по этой кнопке (или нажмите клавишу **F6** клавиатуры. Она дублирует эту кнопку).
После этого, Вы увидите, что строка с командой **goto Start**, которая "дислоцируется" в конце "шапки" программы, выделится:

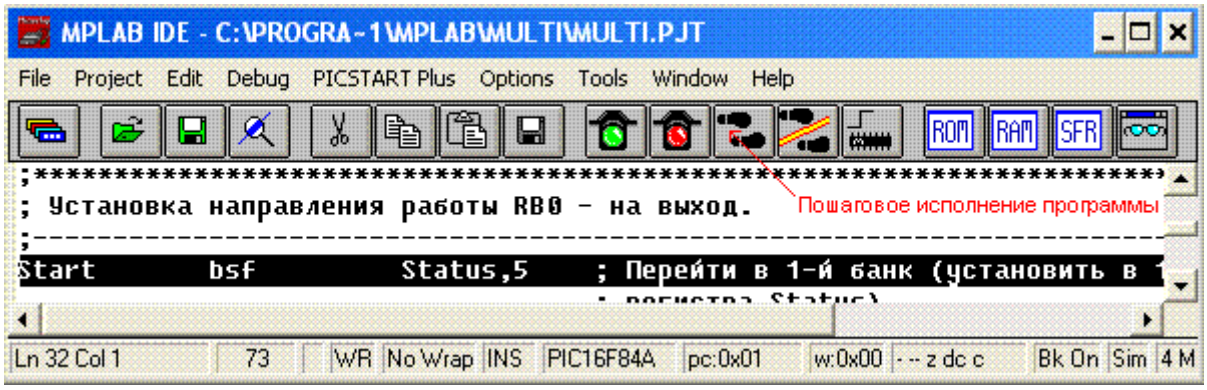


Обращаю Ваше внимание на то, что эта выделенная команда занимает нулевой адрес в памяти программ, в чем Вы можете убедиться, щелкнув по кнопке (пиктограмме) ***ROM***.
После этого, в окне ***Program Memory Window***, Вы увидите, что строка, с командой **goto Start,** тоже выделилась.
Возьмите это на заметку и закройте окно ***Program Memory Window***.
При нажатии на ***кнопку сброса программы на начало***, Вы всегда будете наблюдать такую же "картину".
Больше эта кнопка ничего "делать не может".
Итак, мы "встали" на начало программы и теперь, в симуляторе, ее необходимо исполнить.
Исполнить программу можно или в пошаговом, или в автоматическом режиме (на выбор).

Пошаговый режим исполнения программы ***применяется в тех случаях, когда нужно "капитально" разобраться с последствиями выполнения тех или иных команд (или всех команд) выбранного "сектора обстрела" (участка программы).***
Автоматический режим исполнения программы ***применяется для выхода на начало этого "сектора", с быстрым, но неконтролируемым результатом исполнения всех предшествующих команд программы, а также для определения временных характеристик программы*** (при помощи секундомера. Об этом - ниже).

Слово "быстрое" означает то, что команды выполняются быстро (скорость их выполнения определяется быстродействием Вашего компьютера), но при этом, невозможно визуально отследить текущие изменения содержимого регистров и порядок исполнения команд.
Эти "дела делаются" в пошаговом режиме, не спеша и основательно.
Для того чтобы задать границы, в пределах которых программа исполняется в "автомате" (от чего и до чего?), нужно назначить так называемые **точки остановки**.
Точкой остановки считается, помеченная программистом, строка текста программы, содержащая команду, название подпрограммы, метку.
Как только "дело доходит" до исполнения команды помеченной строки, автоматическое исполнение программы прекращается (остановка).
Теория теорией, но лучше поближе к практике.

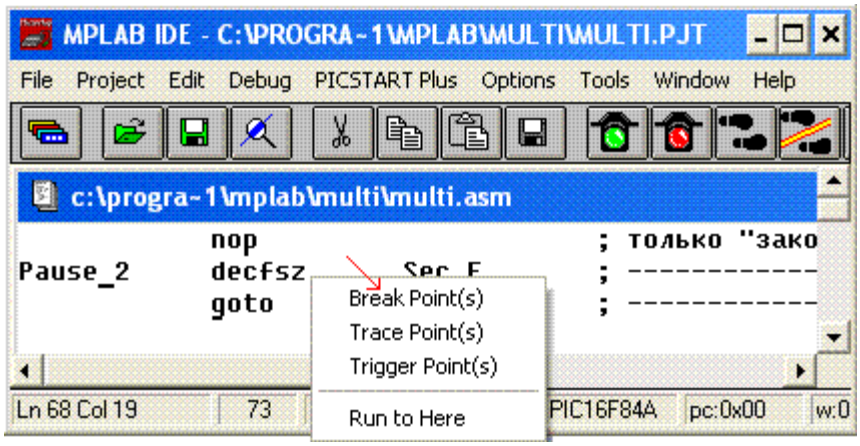
Начну с **пошагового режима**.
Орган управления - ***кнопка пошагового исполнения программы, с нарисованными на ней следами*** (третья слева) **или кнопка F7 клавиатуры.**
Итак, после нажатия на кнопку сброса, рабочая точка программы установилась на начало программы (**goto Start**).
Один раз щелкните по ***кнопке пошагового исполнения программы***.
В соответствии с исполненной командой **goto Start,** произойдет переход на начало ПП **Start**:



Щелкните еще раз.
Выделится строка следующей команды и т.д.
А теперь пощелкайте по ***"следам"*** (или понажимайте кнопку **F7** клавиатуры, что одно и то же) "от души".
Вы увидите порядок исполнения программы.
По ходу работы, Вы можете надолго задуматься над "вскрывшимися обстоятельствами" и никто Вас подгонять не будет. Думайте хоть целый день.
В любой момент, Вы можете остановить/продолжить исполнение программы или нажать на кнопку сброса и снова начать пошаговое исполнение программы.
Хоть сто раз подряд. Вы – "полный хозяин и диктуете свои условия".
Пока "вживитесь в механизм" пошагового исполнения программы, а с деталями того, что при этом будет происходить, разберемся позднее.
В процессе этого "вживления" (лучше "вживления", чем "вживания"), у Вас, естественно, возникнут вопросы, ответы на которые я постараюсь дать.

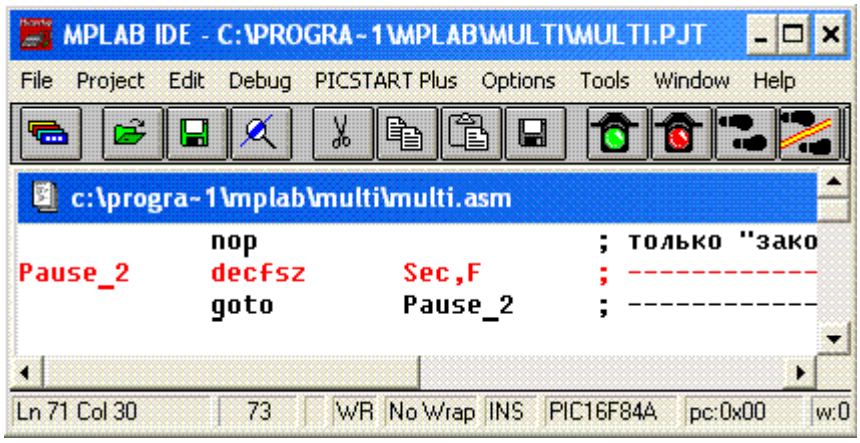
Автоматический режим исполнения программы

Прежде чем работать в автоматическом режиме (в "автомате"), необходимо **назначить точку** (точки) **остановки**.
Если этого не сделать, то программа будет исполняться "вечно" (симулятор "зависнет", то есть, остановки не будет).
В простейшем случае (в основном, так и делается), точку остановки можно назначить прямо в тексте программы (в текстовом редакторе).
Делается это так: щёлкните правой кнопкой мыши по строке подпрограммы **Pause_2**, например, с командой **decfsz Sec,F** (можно щёлкнуть по самой команде, можно по названию подпрограммы, а можно и по пустому месту, короче, в любом месте выбранной строки, до точки с запятой).
Появится окно со списком из четырех позиций:

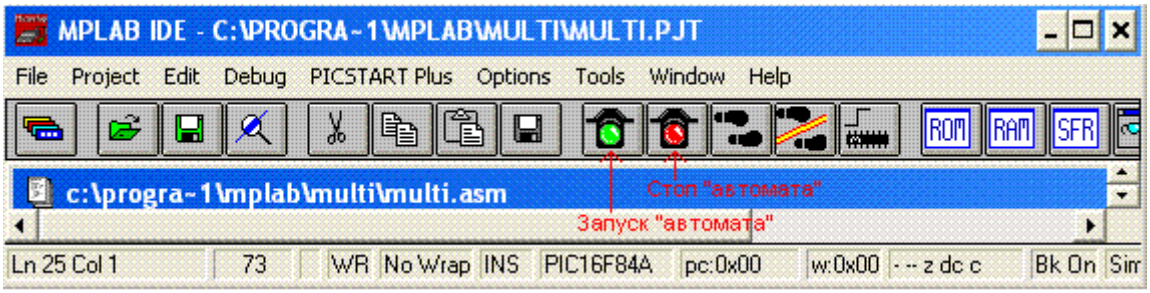


Далее, левой кнопкой мыши, щёлкаете по самой верхней строке в этом списке (***Break Point(s)*** – назначение точки остановки).

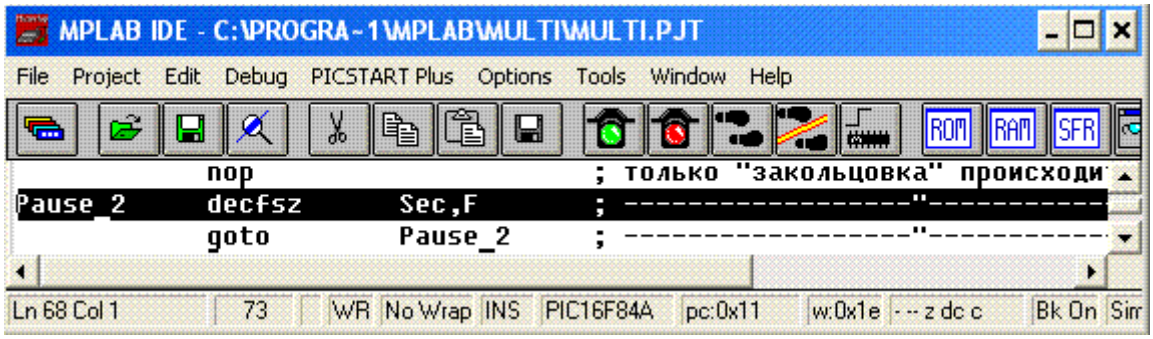
Список "исчезнет", и все что находится в этой строке, выделится **красным цветом** ("индикатор" точки остановки).



Всё. Точка остановки выставлена, и Вы с комфортом можете ее наблюдать в тексте программы.
Сбросьте программу на начало (см. выше) и для того, чтобы запустить режим автоматического исполнения программы (берите на заметку), щёлкните по кнопке с **зеленым** светофором:



Запуск "автомата" можно произвести, если нажать на кнопку **F9** клавиатуры.
На время отработки "автомата", самая нижняя строка окна **MPLAB** (в ее начале стоит **Ln...**) окрашивается в **желтый** цвет.
В данном случае, исполнение программы происходит так быстро, что момент окрашивания можно и не заметить (быстро промелькнет). Учтите это.
Вообще же, эта **желтая** полоса является, как бы, индикатором рабочего процесса автоматического исполнения программы в симуляторе, и по ней можно судить, "зависла" ли программа (**желтое** выделение долго не снимается) или нет.
В нашем случае, "зависания" нет, и программа, в симуляторе, будет исполнена с ее начала и до назначенной точки остановки, что и наблюдается.
То есть, рабочая точка программы остановилась на ранее указанной команде (на точке остановки).

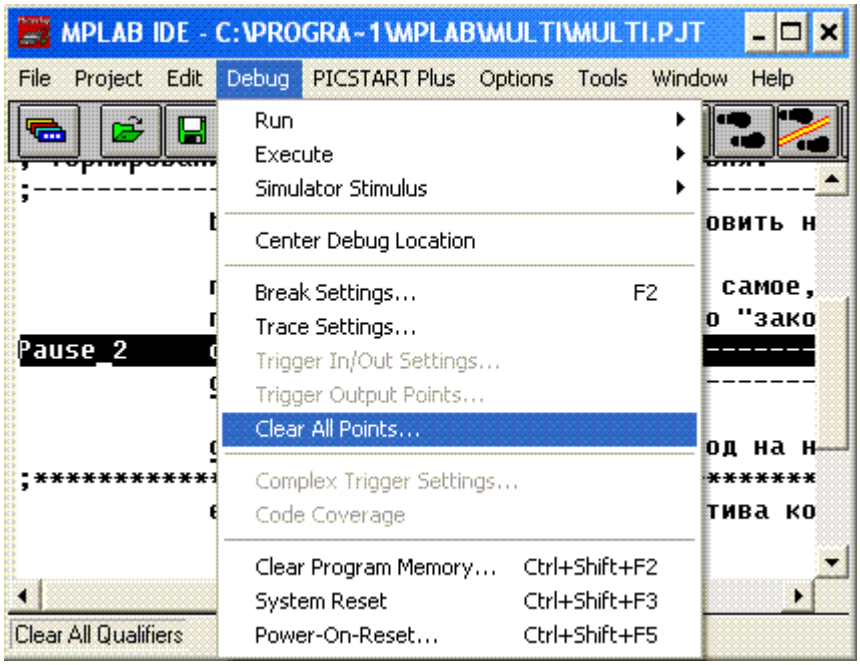


Потренируйтесь.

Несколько раз сбросьте и запустите автоматическое исполнение программы.
Зачем этот "автомат" нужен? В чем преимущества?
Предположим, из всей программы, на данном этапе отладки, необходимо произвести отладку подпрограммы **Pause_2**.
Если работа происходит в пошаговом режиме, то для того чтобы "добраться" до ПП **Pause_2**, придется много раз щёлкать по кнопке (или нажимать на клавишу), что, согласитесь со мной, не совсем удобно. В "автомате", это делается гораздо быстрее. Вот Вам и ответ.
Еще раз напоминаю, что в "автомате", программист не имеет возможности контролировать содержимое регистров. Эти "дела" делаются при пошаговом исполнении программы.
Далее, можно капитально "просканировать" работу ПП **Pause_2** в пошаговом режиме.
Если после этого необходимо перейти к отладке, например, ПП **Pause_1**, то нужно снять (или не снимать, если она потребуется в дальнейшем) "старую" точку остановки и назначить "новую".

Снятие точки остановки.

В главном меню **MPLAB**, щелкните по слову ***Debug***, а затем, в выпадающем списке, по строке ***Clear All Points*** (снять все точки остановки):

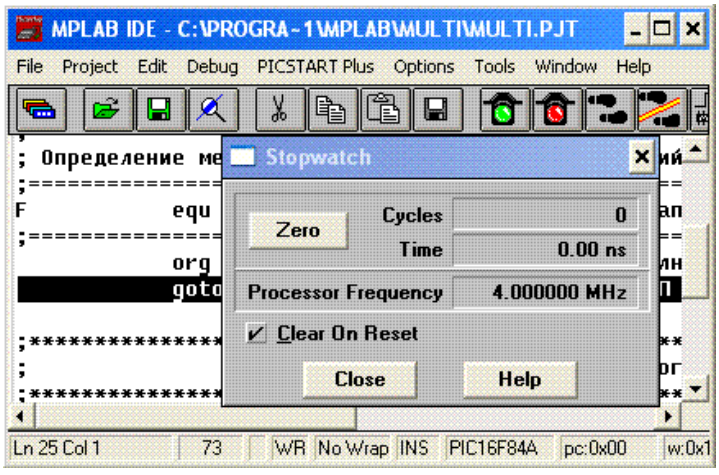


Появится окно с вопросом: ***"Вы действительно хотите удалить все точки остановки?"*** Щелкайте по ***Да***.
После этого, со всех точек остановок снимается выделение **красным цветом** (точек остановок нет).
Еще один способ установки точек остановки:
В главном меню **MPLAB**, щёлкните по слову ***Debug***, а затем по строке ***Break Settinds***.
Откроется окно ***Break Point Settings***.
В списке этого окна, Вы увидите содержимое первого столбца текста программы, с которой происходит работа.
Этот способ установки точек остановки удобен тем, что можно заранее указать, на какой из подпрограмм или меток нужно остановиться, а затем включать или выключать те или иные точки остановки по своему усмотрению.
Но в этом случае, в отличие от описанного выше способа, нельзя выставить точку остановки на строке программы, не содержащей названия подпрограммы или метки.
В окне ***Break Point Settings***, щёлкните по кнопке окошка **Start**.
Выпадет список. Щелкните, например, по строке **Pause_1**.
Название **Pause_1** "ушло" в окошко.
То же самое проделайте и с окошком **End**.
Щелкните по кнопке ***Add***.
В большом списке окна ***Break Point Settings*** появится строка с галочкой.
Если галочка установлена, то точка остановки активна, если не установлена, то пассивна.

Включение и выключение происходит при щелчке по квадратику с галочкой.
При снятии точек остановки (***Debug → Clear All Points***), снимается только галочка.
Чтобы, после этого, снова установить точку (точки) остановки, необходимо открыть окно ***Break Point Settings*** (см. выше) и установить в нем галочку (галочки).
Для того чтобы удалить строку с точкой остановки из большого списка, необходимо ее выделить, а затем щелкнуть по кнопке ***Remove***.
Большой список может содержать несколько точек остановок (вплоть до суммарного количества подпрограмм и меток).
Кнопка с красным светофором (или клавиша **F5** клавиатуры. Она ее дублирует) необходима для ручной остановки "автомата", и соответственно, пользоваться ей можно только в "автомате".
Если программа не работает так, как нужно, "зависнув" внутри какой-нибудь подпрограммы (**желтое** выделение долго не снимается), то несколько раз остановив и запустив "автомат", можно достаточно точно определить, на каком именно участке программы произошло "зависание".
При работе в "автомате", если точек остановок не назначено, то в симуляторе "зависнет" любая программа.
Можете это проверить на практике, используя программу **Multi.asm**.
Работоспособная программа (или подпрограмма) может "зависнуть" в симуляторе, если ее исполнение зависит от каких-то внешних (по отношению к микроконтроллеру) сигналов, а в симуляторе, эти сигналы, по ходу исполнения программы (или подпрограммы), не сымитированы.
Такого рода имитация (**MPLAB** и это может) - предмет отдельного разговора, а пока просто примите сказанное к сведению.

Секундомер.

Вспомните, как подсчитывалось время отработки задержки в программе **Multi.asm**.
С учетом простоты этой программы, подобные расчеты, конечно же, можно произвести и на бумаге.
А если подпрограмма задержки имеет в своем составе "массивные врезки"?
В этом случае, можно просто "утонуть" в громоздких и сложных расчетах.
Так как **MPLAB** для того и создана, чтобы облегчить труд программиста, в своем составе, она имеет специальную утилиту, которая называется "секундомером".
В ходе исполнения программы в симуляторе (в любом режиме), секундомер считает не только количество машинных циклов, но и время.
В любой момент, программист может сбросить показания секундомера на ноль и начать новый отсчет времени.
Для вызова окна секундомера, нужно, в главном меню **MPLAB**, щёлкнуть по слову ***Window***, а затем, в выпадающем списке, по строке ***Stopwatch***.
Откроется окно ***Stopwatch***. Это и есть секундомер:



В нем Вы увидите установленную ранее частоту кварца, кнопку ***Zero*** (сброс секундомера на ноль) и квадратик с надписью ***Clear On Reset***, в котором можно либо поставить, либо не поставить галочку.

Если галочка установлена, то при переходе на начало программы (сбросе программы), секундомер будет автоматически сбрасываться в ноль, а если не установлена, то такого сброса не будет.
Лично у меня, она установлена, что советую и Вам.
Сбросьте программу **Multi.asm** на начало.
Несколько раз пощёлкайте по кнопке со следами ног (или понажимайте клавишу **F7**), и Вы увидите, что секундомер подсчитывает количество машинных циклов исполнения программы и переводит их в микросекунды.
Примечание: обозначение микросекунды выглядит как **us** потому, что в английском алфавите нет буквы "мю" (эта буква - греческая), а английская буква **u** наиболее к ней близка по написанию.
Секундомер "все посчитает в лучшем виде", с учетом самых сложных "кульбитов", которые "проделывает" рабочая точка программы.
В конечном итоге, он выдаст точный результат подсчета времени исполнения команд на том участке программы, на котором Вы работаете.
Практический вывод из этого следующий: если программисту необходимо подобрать значение константы (констант) для формирования какого-то калиброванного интервала времени, то ему вовсе не обязательно скрупулезно подсчитывать на бумаге количество машинных циклов исполнения команд, а достаточно "грубой прикидки".
"Доводка до нормы" производится путем изменения, в небольших пределах, значения константы (констант), а если этого не достаточно, то и установкой того или иного количества **NOP**ов, с постоянным контролем результатов этих изменений по секундомеру, до получения нужного результата.
Сам по себе, этот процесс довольно-таки не сложен (нужно только "руку набить").
Таким немудрёным образом, можно, с минимальными трудозатратами, отстроить временные характеристики даже очень "навороченных" программ.
Если имеется "чужая" программа и Вам необходимо внести свои коррективы в ее временные характеристики, то это тоже делается при помощи секундомера.
И т.д. Вариаций много.
Естественно, что в таких случаях, без "автомата" не обойтись.
На практике, результат применения такого метода проб и ошибок (с использованием секундомера), часто оказывается гораздо более эффективным, чем точные вычисления на бумаге, но "сбрасывать их со счетов" не нужно.
В идеале, нужен комплексный подход.

6. Что дальше?

То, что Вы узнали (а я надеюсь, что и усвоили), изучая предыдущие разделы, есть основа "прожиточного минимума" программиста ПИКов, необходимая для того, чтобы иметь первичное представление о том, "с какого бока подойти к этой корове и за что ее дергать, чтобы она дала молоко".
В соответствии с ранее сформулированной тактикой обучения, я "оставил за скобками" то, без чего, на первых порах, можно обойтись.
В дальнейшем это будет наверстано.
Если Вы хорошо потрудились, то к настоящему времени, у Вас должна сложиться более - менее целостная картина предмета обучения. Хотя бы в общем виде.
В настоящее время, Ваши знания в области программирования ПИКов, вернее всего, технологичны и представляют собой набор достаточно разобщенных сведений.
Это вполне естественно и нормально.
Поздравляю Вас с преодолением самой нудной и не слишком интересной части работы (естественно, при условии, что Вы хорошо потрудились).
На данный момент, имеется все необходимое для перехода к "плотной" работе с текстами программ, успешное завершение которой будет способствовать "превращению" разобщенного в единое целое.
В этом отношении, ничего лучшего, чем пример поэтапной разработки конкретного устройства на ПИКе, не существует.
В конечном итоге, я, на Ваших глазах, поэтапно "разложу на молекулы" весь процесс конструирования достаточно сложного и "серьезного" устройства.
Причем, это будет сделано нестандартно.
Обычно, конструкторы публикуют конечный, "отлакированный" результат своего труда, но при этом теряется информация о "кухне" конструирования.
А ведь именно это и есть самое важное в процессе конструирования!!!
Мало того, я сознательно усложняю (для себя) задачу.
А именно, устройство будет создано на базе достаточно сложного, современного, электронного компонента, принцип работы которого изначально не известен.
Проще говоря, я поэтапно и с детальным объяснением всех "телодвижений", постараюсь провести Вас по всей "цепочке", от возникновения идеи и до ее практической реализации.
Подобного рода информация является исключительной редкостью по той причине, что существуют весьма значительные трудности, связанные с ее "извлечением из глубин" сознания и подсознания, и с "удобоваримым переносом этой информации на бумагу".
Тем не менее, на свой страх и риск, я проделаю эту работу в надежде на то, что "мой скорбный труд" реально поможет людям, для которых конструирование электронных устройств является либо любимым занятием, либо жизненноважной необходимостью, либо и тем, и другим.
При выборе "предметов объяснения", передо мной встала сложная задача: каким образом сделать так, чтобы "угодить" как можно большему количеству людей?
Ведь электроника многообразна: одному нужно одно, а другому другое, и описать всё и вся невозможно по определению.
Я исхожу из того, что существуют устройства, которые совмещают в себе разнообразные базовые функции, и если на примере одного из таких устройств, подробно объяснить, как эти базовые функции реализуются на практике, то в процессе конструирования, такого рода знания могут быть использованы для решения широчайшего класса задач.
В качестве такого устройства я выбрал 3-х диапазонный частотомер с функцией цифровой шкалы.
Может быть, кого-то из Вас и не устроит такой выбор, но суть предлагаемого состоит совсем не в этом конкретном выборе (можно выбрать и что-то другое), а в том, что по ходу объяснений, будут формулироваться общие принципы организации тех или иных процедур, с дальнейшим их "разложением на молекулы".
Все это будет "привязано" к последовательному и логически мотивированному объяснению работы конкретного устройства, которое, в конечном итоге, Вы самостоятельно сможете изготовить.
На мой взгляд, это самый эффективный способ достижения поставленной цели.
"Въезд" в конструирование устройств на м/контроллерах представляет собой непрерывную цепочку последовательных действий, разрывать которую не рекомендуется (получите

"штрафные круги", как в биатлоне).
Можете это рассматривать как мой призыв к Вашему здравомыслию и прагматизму.
"Лезть на Эверест" без специальной подготовки ("прыгать через две ступеньки") не стОит, так как ничего хорошего из этого не выйдет.
Сначала нужно получить "базовые" знания и навыки, а только после этого "выходить на "охоту в страну непуганых зверей", а то ведь "задерут как пить дать" ("звери" там серьезные и совсем без чувства сострадания и юмора).
Понимаю, что это и трудно, и хлопотно, и терпения нужен "целый вагон", но это и есть самый короткий и эффективный путь.
Более эффективного варианта "въезда" просто не существует (Вы уж поверьте человеку, которого собственная бестолковость многократно "водила носом по батарее").
Надеюсь, что я Вас убедил.
Так что, давайте создавать "базу".
Начинаем "вгрызаться" в то, что имеем.
Для начала, я объясню Вам детали процесса конструирования какого-нибудь простого устройства.
Если Вы думаете, что далее последует что-то типа "сверления зуба", то Вы ошибаетесь и будете приятно удивлены тем, что речь пойдет не о "страшном и ужасном".

Из личного опыта:
При "въезде" в м/процессорную технику, на мой взгляд, существует некая, достаточно четкая, "граница" (сужу по своим ощущениям).
До нее → "каша в мозгах, шизофрения, битье головой о стену" и прочие гадости.
После нее → что-то типа "просветления"/"нирваны" ("всё становится на свои места").
В том смысле, что происходит "чудесный въезд" в "глобальную" стратегию конструирования устройств на м/контроллерах: человек начинает понимать истинный смысл и предназначение того, что раньше было "темным лесом", и самое главное, приобретает способность свободно манипулировать этим "оружием" в своих целях.
Остальное (тактика), как говорится, дело наживное.
Все это можно отнести к "проделкам" подсознания, но ведь "эти проделки на голом месте не рождаются и с неба не падают".
Чтобы подсознание преподнесло человеку такой приятный "сюрприз", нужно хорошенько поработать со своим сознанием, и не примитивно, как отличник, которому нужно заработать пятерки для получения золотой медали, а живя этим занятием.
Лично у меня, критерий "включения" подсознания такой: если снится работа над программой (не важно какая, пусть даже самая идиотская), значит все в полном порядке.
У разных людей по-разному, но так или иначе, то что я назвал "границей", реально существует.
Желаю Вам ее "перейти".

7. Пример создания программы (начало).

Общие положения

Порядок разработки устройств на микроконтроллерах следующий:
Предположим, что возникла идея создания устройства на микроконтроллере, которая "упирается" в создание программы, обеспечивающей реализацию замысла конструктора.
Программа создается не сама для себя, а под конкретное устройство.
Цель программы - обеспечения "нормального" (в соответствии с задуманным) функционирования этого устройства.
Значительная часть электронной начинки задуманного устройства (кроме внешних, по отношению к микроконтроллеру, устройств) находится внутри микроконтроллера.
В соответствии с логикой (алгоритмом) работы программы, заданной программистом, программа формирует состав этой электронной начинки (что-то включает, что-то выключает) и обеспечивает логическое взаимодействие ее составных частей.
Специфика конструирования устройств на микроконтроллерах заключается в том, что в большинстве случаев, наличия отработанной блок - схемы программы оказывается достаточным для создания его полной принципиальной схемы или той части принципиальной схемы, которая включает в себя микроконтроллер.
И в самом деле, если осуществлена "жесткая привязка" внешних (по отношению к микроконтроллеру) цепей и устройств к конкретным выводам микроконтроллера (функции выводов определены), типы внешних цепей и устройств определены и "привязаны" к рабочим уровням напряжения на выводах микроконтроллера, то он представляет собой **базовый блок** с "четко" определенными функциями, к выводам которого подключаются входы (управление внешними устройствами) и выходы (управление базовым блоком) внешних устройств.
В процессе "доведения аппаратуры до ума", при неизменных функциях базового блока, внешние устройства могут видоизменяться, заменяться на аналогичные и т.д.
Именно к такого рода определенности и нужно стремиться при конструировании устройств на микроконтроллерах.
Естественно, что в процессе работы, могут производиться корректировки/изменения (например, переназначение функций выводов портов, изменение логики управления внешними устройствами и т.д.).
Процесс создания "электронной начинки" устройства можно разделить на два этапа, которые характеризуются словами "грубо" и "точно" (по аналогии со стратегией и тактикой).
"Грубый" этап включает в себя создание блок-схемы устройства, в которой "центром вселенной" является базовый блок на микроконтроллере.
Блок-схема устройства состоит из блок-схемы программы (основная "масса") и "квадратиков" (условно), обозначающих внешние устройства.
На этом этапе работы, нужно ясно и четко представлять себе функции, выполняемые той или иной составной частью блок - схемы, а также и порядок их взаимодействия.
Например, если в блок - схему программы введен счетчик, то как минимум, необходимо знать правильный ответ на вопрос: "Зачем он вообще нужен и каковы его функции в "общей картине" устройства"?
Если без него обойтись нельзя, то в контексте задумки, нужно определиться с его параметрами (время и направление счета, активный фронт и т.д.).
Это потребуется при составлении программы.
Естественно, что необходимо знать принципы работы устройств цифровой техники (а часто и аналоговой), а иначе будет "ёжик в тумане".
По этой причине, "ставки" начинающих программистов, имеющих опыт практической работы в области цифровой и аналоговой техники, выше "ставок" начинающих программистов, которые этого опыта не имеют, хотя, в конечном итоге, результат работы зависит от трудолюбия и "упёртости".
После того, как определены элементы блок - схемы программы и порядок их взаимодействия, определяются функции выводов микроконтроллера.
То есть, в соответствии с функциональным предназначением выводов м/контроллера, определяется порядок подключения, к ним, входов и выходов внешних устройств.
В дальнейшем, при составлении программы, этот порядок подключения будет определять содержание команд, производящих действия с данными, поступающими в микроконтроллер от внешних устройств, и с данными, которые формируются микроконтроллером для

управления внешними устройствами.
После этого, сначала, составляется принципиальная схема базового блока (рисуется м/контроллер), а потом, к выводам микроконтроллера, определенным ранее, "подключаются" входы или выходы внешних устройств.
Если конструктор еще не определился с принципиальными схемами внешних устройств, то их можно изобразить в виде "квадратиков".
Если конструктор заранее определился с принципиальными схемами внешних устройств, то можно составить полную, принципиальную схему разрабатываемого устройства и "привязать" программу к специфике работы внешних устройств.
Второй вариант, на мой взгляд, более предпочтителен, так как он предполагает изначальную продуманность конструкции устройства и соответственно, меньшее количество непредвиденных "сюрпризов", хотя и первый вариант не плох, особенно для программистов с опытом работы.
На этом, "грубый" этап работы заканчивается и начинается "точный".
Итак, принцип работы устройства (стратегия), в основном, продуман и понятен.
Принципиальная схема в наличии.
Теперь можно начинать составление программы.
До этого предполагалось, что базовый блок работает "идеально" (без ошибок).
Теперь нужно, программными и аппаратными средствами ПИКа, обеспечить это на практике.
Работа по составлению программы начинается с "шапки" программы, в которой нужно "прописать" все регистры специального и общего назначения, которые предполагается использовать в программе.
Если Вы что-то забудете "прописать", а это "что-то" задействуется в рабочей части программы, то ничего страшного, **MPLAB** Вам об этом обязательно напомнит.
По ходу создания программы, часто появляется необходимость "прописки" дополнительных регистров.
Проблем в этом никаких нет. Они "прописываются" так же, как и ранее "прописанные" регистры и тут же "вводятся в бой".
Основные "сражения с непослушными нулями и единицами развернутся" в рабочей части программы.
Здесь будут "стрелять пушки, бить барабаны, свистеть пули и литься кровь".
Желаю Вам оказаться победителем в этом "сражении" и испытать радость победы над самим собой.
Обычно, рабочая часть программы начинается с **подготовительных операций**.
И в самом деле, принципиальная схема устройства в наличии, а значит можно осуществить "привязку" программы к внешним цепям и устройствам.
Например, если к выводу **RB0** порта B подключен светодиод, а к выводу **RA4** порта A подключен выход, например, формирователя импульсов, то в начале программы (в ПП **Start**), вывод **RB0** необходимо переключить на работу "на выход", а вывод **RA4** переключить на работу "на вход".
Таким образом, речь идет о начальной "настройке" битов каких-то из регистров специального назначения (в данном случае, **TrisA** и **TrisB**).
Очень часто требуется сбросить в ноль содержимое каких-то из регистров общего назначения (подготовить их к заполнению данными).
Или еще что-нибудь. Зависит от конкретной программы.
Чем сложнее программа, тем "массивнее" подготовительные операции.
А вот после подготовительных операций, начинается то, что описать, прямо скажу, сложновато.
Да и как можно описать все детали игры, например, в шахматы?
Возможных комбинаций – миллионы. Жизни не хватит.
Тема → неисчерпаемая.
Для людей творческих и ищущих → настоящий "Клондайк", где можно развернуться "во всю ширь" и по максимуму реализовать свой творческий потенциал.
Естественно, что свою задачу я вижу не в том, чтобы "перелопачивать" всё многообразие различных решений, а в том, чтобы помочь Вам приобщиться к этому очень перспективному и увлекательному занятию, а заодно и помочь Вам стать "жизнеспособным организмом", склонным к самостоятельному развитию (самообразованию).
Если быть объективным, то заголовок этого раздела не в полной мере отражает суть того, чем мы будем далее заниматься.

Необходимо уточнить: мы будем заниматься конструированием устройств на ПИКах.
Для того чтобы это сделать, нужен "симбиоз" электронщика и программиста.
"Однобокость" типа "или электронщик, или программист" ни к чему хорошему не приведет (машина без колес или колеса без машины).
Учтите это.
Так как "Самоучитель..." рассчитан, в первую очередь, на начинающих, то устройства, программы под них или фрагменты программ, изначально, будут простыми.
В дальнейшем, их сложность будет постепенно нарастать.
Все файлы текстов программ или их частей, будут иметь расширение **.ASM** и их можно будет открыть в **MPLAB**.

Подробное описание процесса конструирования устройства на PIC16F84A.

Примечание: все, что будет описываться ниже, для меня уже пройденный этап и "логические цепочки" я, естественно, буду восстанавливать по памяти.
Постараюсь это сделать в удобной для восприятия форме.
Примеры взяты из реальной жизни (моей).
Описываемые ниже устройства, работоспособны, выполняют свои функции и проверены в работе.

Пример

В большинстве случаев (но не всегда), все начинается с того, что возникает некая необходимость.
В свое время, я работал в системе МЧС Липецкой области и занимался вопросами технического обслуживания, ремонта средств радиосвязи и модернизации существующей системы радиосвязи.
Возникла необходимость в создании относительно дешевого, двунаправленного ретранслятора на импортных симплексных радиостанциях Vertex - 2000, а также в переводе однонаправленного ретранслятора Vertex - 7000 в двунаправленный режим.
Эти задачи были успешно решены при помощи простых (в схемотехническом отношении) устройств на **PIC16F84A**, которые обошлись очень дешево, но тем не менее, сэкономили "кругленькую сумму", время и нервы.
Для начала, рассмотрим самое простое из этих устройств - устройство формирования сигнала тонального вызова для оконечной аппаратуры управления ретранслятором, в качестве которой использована радиостанция Vertex - 2000.
Начинаем "раскрутку".
Суть проблемы:
Система тонального вызова необходима для вызова корреспондента в том случае, если его трубка уложена в специально отведенное для этого место.
В таком положении, в режиме дежурного приема, радиостанция "не слышит" ничего, кроме строго определенного по частоте сигнала тонального вызова и "докричаться" до корреспондента, с уложенной трубкой, можно только послав ему сигнал тонального вызова.
Радиостанция Vertex - 2000 имеет систему тонального вызова, но ни одно из значений частот тонального вызова, которые предлагаются разработчиками и которые можно установить при программировании радиостанции, не соответствует значению частоты тонального вызова, применяемой в системе "МЧС-радиосвязи" Липецкой области.
Самая близкая, к этому значению, частота тонального вызова, которую можно установить в Вертексе, не попадает в частотный интервал допуска.
Так как в Вертексе, возможности коррекции пользователем частоты тонального вызова не предусмотрено (можно только выбрать из списка), то возникает необходимость в создании устройства формирования сигнала тонального вызова с высокостабильной частотой **1450гц**.
В простейшем случае, управление работой этого устройства можно сделать ручным (при помощи кнопки выкл./выкл. тонального вызова), но это уже "прошлый век" и не совсем удобно для пользователя.
Кроме того, в этом случае, время "выдачи" в эфир сигнала тонального вызова нестабильно и существует вероятность "недодачи" (по времени) этого сигнала, при которой приемник тонального вызова корреспондента, из-за своей инерционности, просто "не успеет" сработать, да и "перебор" тоже не нужен.
Следовательно, при включении на передачу, необходимо организовать автоматический

режим "выдачи" в эфир сигнала тонального вызова в течение калиброванного интервала времени.
Так как для пользователя не всегда бывает удобным "запуск" устройства формирования тонального вызова каждый раз, когда происходит включение на передачу, то необходимо сделать так, чтобы режим автоматического, тонального вызова можно было бы включить или отключить.
В итоге, к устройству формирования сигнала тонального вызова предъявляются следующие **основные требования**:

1. **Частота сигнала тонального вызова должна быть высокостабильной и равной 1450гц.**
2. **Сигнал тонального вызова должен формироваться при каждом нажатии тангенты микрофонной гарнитуры.**
3. **После нажатия на тангенту, сигнал тонального вызова должен "выдаваться" в эфир в течение калиброванного интервала времени (оптимально подобранного по длительности).**
4. **Режим тонального вызова должен включаться и отключаться вручную.**

Примечание: естественно, что первые 3 пункта должны выполняться только при включенном режиме тонального вызова.
Итак, "глобальные" требования сформулированы в конкретной и однозначно понимаемой форме. Теперь начинаем "раскрутку" по пунктам.

Первый пункт

Если речь идет о высокостабильной частоте, то тактовый генератор ПИКа должен быть кварцевым (если не требуется высокой стабильности частоты, то его можно перевести в режим RC-генератора. В ПИКах и такое предусмотрено).
В этом отношении, "изобретать велосипед" не нужно. Будем использовать стандарт: стандартный кварцевый генератор (называется **ХТ**) с кварцем на 4 мГц.
В этом случае, один машинный цикл равен одной микросекунде, что очень удобно при расчетах и "прикидках".
При использовании кварца, время "выдачи" в эфир сигнала тонального вызова также окажется высокостабильным, хотя к этому интервалу времени, высоких требований по стабильности не предъявляется.
Как говорится, "кашу маслом не испортишь". Будем считать это приятной неожиданностью.
В принципе, можно сформировать импульсную последовательность любой скважности.
Исходя из того, что анализатор приемника тонального вызова корреспондента представляет собой цифровое устройство, некритичное к скважности импульсной последовательности, вырабатываемой на передающей стороне, то каких-то жестких условий по скважности нет, за исключением того, что импульсы должны быть "не слишком короткими и не слишком длинными" (учет инерционности радиотракта).
"Не мудрствуя лукаво", задаем отношение длительности периода к длительности импульса = 2 (так называемый "меандр").
В части касающейся процесса формирования периода импульсной последовательности сигнала тонального вызова, тоже мудрить не нужно: полупериоды формируем, используя закольцовку рабочей точки программы в подпрограммах задержки (по аналогии с программой **Multi.asm**).
В части касающейся 1-го пункта, всё, что возможно определить на предварительной стадии разработки устройства ("грубая прикидка"), определено.
К моменту начала составления текста программы, желательно определиться с как можно бОльшим количеством исходных данных.
Такая продуманность имеет наибольший, положительный эффект.

Второй и третий пункты

Так как требования 2-го и 3-го пунктов между собой связаны, то для их реализации, определение исходных данных лучше всего производить в комплексе.
Совершенно очевидно, что при нажатии на тангенту (включении на передачу), должен быть запущен какой-то механизм, формирующий сам сигнал тонального вызова и "дозирующий", по времени, его "выдачу" в эфир.
Так как программа должна исполняться непрерывно, то в случае нахождения радиостанции в режиме дежурного приема, программа должна "закольцеваться" (рабочая точка программы, до последующего нажатия на тангенту, должна, в какой-нибудь подпрограмме, "уйти в вечное кольцо").

Часто такого рода "закольцовки" происходят в начале программы, а именно, в подпрограмме **Start**, но можно "закольцевать" программу (вернее, ее рабочую точку) и в другом "месте" программы.
Опять же, не будем "изобретать велосипед".
В такой простой программе, "особо не развыбираешься".
Итак, в режиме дежурного приема, рабочая точка программы "закольцовывается" в ПП **Start** и не выходит из нее до момента возникновения управляющего сигнала включения на передачу.
Примечание: это то, что я называю **"вечным кольцом"**. Может быть, это определение и не вполне удачно, но оно короткое и отражает суть происходящего.
В момент прихода управляющего сигнала, рабочая точка программы должна выйти из этого "вечного кольца" по сценарию "программа исполняется далее".
А теперь прикинем, а что же должно быть далее?
Естественно, должен начаться процесс формирования одного периода тонального сигнала, соответствующего частоте 1450гц.
Предположим, мы его сформировали, а дальше?
Совершенно очевидно, что необходима "закольцовка" рабочей точки программы, с конца, на начало процедуры формирования одного периода, а иначе сформируется один период и на этом все закончится.
Если поставить время нахождения, в этой "закольцовке", в зависимость только от времени нажатия тангенты, то тональный сигнал будет "выдаваться" в эфир в течении всего времени нажатия тангенты, что неприемлемо (см. выше).
После нажатия тангенты, тональный сигнал должен "выдаваться" в эфир в течение **3-х сек**.
Примечание: давайте, для определенности, "привяжемся" к этому значению. Такое значение является оптимальным и нужно ориентироваться на него.
То есть, процесс формирования сигнала тонального вызова должен быть поставлен в зависимость не только от сигнала включения на передачу, но и от таймера (счетчика времени), отмеряющего, с момента нажатия на тангенту, 3 секунды.
Таймер должен что-то считать.
Решение напрашивается само собой: считать нужно количество периодов сигнала тонального вызова.
Записывая в таймер различные значения констант, можно получить те или иные значения времени "выхода" в эфир сигнала тонального вызова, а следовательно, и "отрегулировать" это время.
Таким образом, можно сформировать следующий **принцип работы программы:**
В режиме дежурного приема, в подпрограмме **Start**, рабочая точка программы должна "уйти в вечное кольцо" и "мотать в ней кольца" до момента переключения радиостанции с приема на передачу.
После того как это случится, рабочая точка программы должна выйти из этого "вечного кольца" по сценарию "программа исполняется далее", и еще раз "закольцеваться" в процедуре формирования периода тонального сигнала.
Одновременно должен начать работать таймер, время срабатывания которого зависит от количества периодов тонального сигнала, сформированных программой, с момента нажатия на тангенту.
После отсчитывания заданного количества периодов (через 3 сек.), таймер должен "сработать", и рабочая точка программы, не зависимо от того, находится ли радиостанция на передаче или нет, до следующего рабочего перепада управляющего сигнала (до следующего перехода от приема к передаче), должна снова "уйти в вечное кольцо" подпрограммы **Start**.
Все это, с непривычки, выглядит чем-то не вполне понятным, но после работы с текстом программы, многие из Вас удивятся, как это, в сущности, просто.
Для того чтобы еще более конкретно оформить "конструкцию" программы, необходимо ответить на вопрос: "В каком месте программы должны записываться константы (числа, определяющие величину интервала времени "выхода" тонального сигнала в эфир)"?
Дело в том (а это очень важно и сейчас, и при работе с другими программами), что константы, используемые в работе устройств, подобных упомянутому выше таймеру (устройств, которые что-то подсчитывают), должны "закладываться" (записываться) в эти устройства предварительно, то есть, до начала их работы.
Эта "закладка" должна происходить вне цикла работы этого таймера (счетчика), а иначе, значения констант будут обновляться каждый цикл и таймер никогда не сработает (никогда

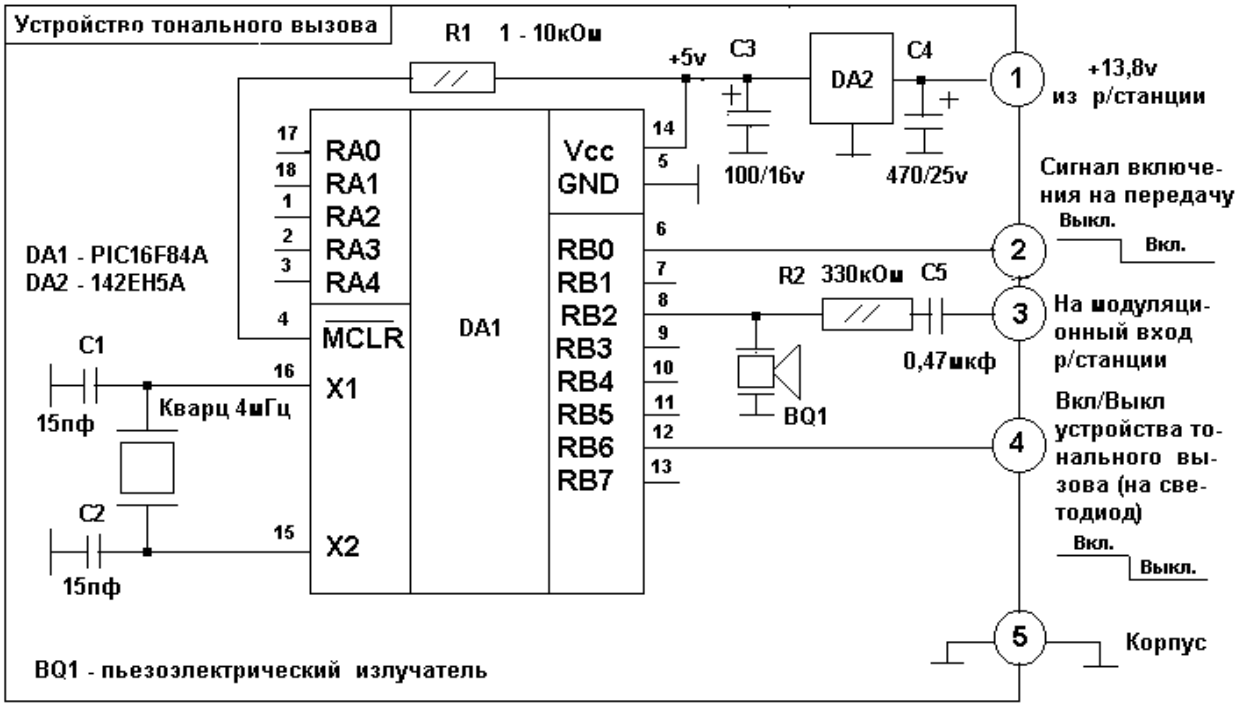
не досчитает до нуля).
В нашем случае, "особо не развыбираешься", так как вне 3-х секундного "кольца" находится только подпрограмма **Start**.
Таким образом, задача сводится к определению "места", внутри ПП **Start**, в которое можно "врезать" команды записи констант.
В частности, их можно "врезать" между командами подготовительных операций, но это затрудняет комфортное восприятие текста программы.
Во избежание этого, лучше всего поместить группу команд записи констант в конце подпрограммы **Start**.
Да будет так.

Четвертый пункт

Для реализации этого пункта, нужен какой-то орган управления типа переключателя или кнопки.
В данном случае, можно задействовать кнопку A (аксессуары) радиостанции Vertex - 2000, расположенную на передней панели радиостанции, что очень удобно.
Она задействуется при программировании радиостанции ("зашивке" частот и режимов), а в рабочем режиме, если при программировании ей не присвоено никаких функций (что и имеет место быть), она просто свободна и ее можно использовать в качестве выключателя (или "включателя") режима тонального вызова.
Если использовать эту кнопку (фиксации нет), то в качестве элемента оперативной памяти, придется назначить дополнительный регистр общего назначения, работающий как триггер (элемент оперативной памяти), состояния которого зависят от состояния кнопки на момент опроса.
Кроме того, если речь идет о более-менее качественном устройстве, то необходимо будет принять меры по защите от дребезга контактов, а если речь идет об очень качественном устройстве, то необходимо будет задействовать энергонезависимую (EEPROM) память данных ПИКа (для энергонезависимого сохранения настроек) или после включения питания, по умолчанию, включать (или выключать) режим тонального вызова.
Все это сделать можно, но если разработчик поставлен перед такого рода выбором, то прежде всего, нужно повнимательнее изучить аппаратуру.
А вдруг в ней имеется то, что нужно?
В данном случае, то что нужно, в наличии имеется, и поэтому просто нет смысла усложнять программу.
И в самом деле, в "Вертексе" имеется и кнопка, и светодиодный индикатор ее включения и выключения.
В нем имеется и оперативная память интересующей нас кнопки, и "противодребезг", и после включения питания р/станции, сигнал управления, формируемый этой кнопкой, представляет собой сигнал управления типа "кнопка отжата".
Для того, чтобы это выяснить, нужно просто понаблюдать за поведением светодиодного индикатора при нажатиях/отжатиях этой кнопки и при коммутациях питания.
К сожалению, EEPROM память данных микропроцессора р/станции под эту кнопку не задействована, но с этим вполне можно мириться.
Для того чтобы "запустить в эксплуатацию" этот "подарок судьбы", нужно всего-лишь сделать отвод от того места пайки выводов светодиода, на котором наблюдается наиболее резкий перепад напряжения и в дальнейшем, "привязать" программу к логике этого перепада.
При этом, никаких преобразователей уровней (для согласования уровней) вводить в схему разрабатываемого устройства не нужно, так как Вертекс собран на пятивольтовых микросхемах.
Переходим к составлению принципиальной схемы.
Так как применяется стандартный кварцевый генератор (**XT**), то можно использовать стандартную схему подключения кварца.
Для того чтобы не использовать сетевой блок питания, можно запитаться от радиостанции. 5 вольт в ней имеется, но лучше излишне не перегружать 5-вольтовый стабилизатор напряжения р/станции и ввести в принципиальную схему устройства свой 5-вольтовый стабилизатор на м/схеме 142ЕН5А, для которой, в связи с малым энергопотреблением, радиатор не нужен.
Сигнальные цепи, по которым будет осуществляться обмен данными между ПИКом и радиостанцией, определены ранее.

Остается только выяснить, какие именно выводы ПИКа задействовать под эти цепи?
Например, назначим следующую "привязку":
Вывод порта B **RB0** будет **входом управления (прием/передача).**
Этот вывод нужно подключить к тому контакту кнопки тангенты, который не соединен с корпусом.
В этом случае, режиму приема будет соответствовать **1**, а режиму передачи **0**.
Вывод **RB2** назначаем **выходом устройства тонального вызова**.
Его нужно подключить к модуляционному входу передатчика.
Так как этот вход имеет высокую чувствительность, то выходной уровень тонального сигнала нужно понизить, и на всякий случай, принять меры "развязки" по постоянному току.
В простейшем случае, это можно сделать, если включить в разрыв цепи, соединяющей выход устройства тонального вызова с модуляционным входом передатчика, последовательную RC-цепочку.
Вывод **RB6** назначаем **входом управления (вкл/выкл режима тонального вызова).**
Этот вывод нужно подключить к тому месту пайки индикаторного светодиода кнопки А радиостанции, на котором, при переключениях этой кнопки, наблюдается максимальный перепад напряжения.
Для этого нужно произвести замер напряжения, после чего определиться с тем, какому именно из двух состояний светодиода ("горит/не горит") будет соответствовать, например, режим включения устройства тонального вызова?
Определим так (но можно и наоборот):
- включению устройства тонального вызова будет соответствовать **1**,
- а выключению **0**.

Таким образом, уровни управляющих сигналов "четко привязались" к исполняемым операциям, и к ним, в дальнейшем, будет "привязана" программа.
Для слухового контроля за процессом "выхода" в эфир сигнала тонального вызова, к выводу **RB2,** можно подключить пьезоэлектрический излучатель.
Теперь ничто не мешает составить принципиальную схему устройства:



Примечание: выводы портов можно "привязать" к сигнальным цепям в какой угодно комбинации. Можно назначить их и в порте А.
По вполне понятным причинам, эту "привязку" нужно осуществить до начала составления программы.
Номинал **R2** нужно подбирать по наилучшему качеству модуляции.
Для р/станций Vertex-2000 он такой, как указано в схеме.
Для других типов радиостанций, возможно, его нужно будет подобрать.

Если ПИК не нужно сбрасывать внешним сигналом сброса, а в данном случае, это так, то между выводом **MCLR** и цепью +5v подключается **R1** номиналом, обычно, от 1 до 10ком.
Можно подключить и напрямую (без резистора), но я, на всякий случай, "подстрахуюсь".
Номиналы конденсаторов **C3** и **C4** указаны с запасом. Можно сделать их и поменьше.
Внешние цепи устройства тонального вызова 1 … 5 подключаются к разъему на задней стенке "Вертекса".
В нем есть незадействованные контакты, на которые можно вывести сигналы, необходимые для его работы (для выводов **RB0** и **RB6**), а также и цепь **+13,8v**.
Вывод корпуса и вывод модуляционного входа имеются.
Разъем монтируется прямо на печатной плате устройства и поэтому, при подключении его к р/станции, никаких механических креплений не нужно.
Итак, своеобразная "болванка" замысла программы создана.
Теперь нужно наполнить ее содержанием.
"Скелет" программы (стратегический замысел) сформирован ранее.
Принципиальная схема есть.
Остается только составить **блок - схему программы**.
Перед ее составлением, нужно определиться с названиями подпрограмм.
Назначаем следующие названия подпрограмм:
Функции подпрограмм **START, PAUSE_1, PAUSE_2** были рассмотрены ранее.
Циклическую подпрограмму формирования времени "выхода" сигнала тонального вызова в эфир назову, например, **CYCLE**, а подпрограмму, в которой осуществляется "уход в вечное кольцо", назову, например, **PRD**.
Можно назвать их и по-другому. Это зависит от фантазии.

Блок-схема (алгоритм) программы

Переходим к тексту программы.
Я сделал два варианта текста программы. Они отличаются только комментариями.
Вариант текста программы, с названием **cus_2.asm,** содержит комментарии в формализованном виде. То есть, они написаны на формальном языке команд.
Вариант текста программы, с названием **cus_1.asm,** содержит неформализованные комментарии, которые объясняют суть производимых действий.
Совместное использования обеих этих текстов должно Вам помочь.

Оба этих ASM-файла прилагаются (см. папку *"Тексты программ"*).

Текст программы **cus_1.asm** выглядит так:

```
;*******************************************************************************
; cus_1.asm                  ВАРИАНТ КОММЕНТАРИЯ № 1
;      Программа разработана для устройства тонального вызова с частотой 1450 Гц.
;*******************************************************************************
; Автор: Корабельников Евгений Александрович  г.Липецк,  январь 2005г.
; E-mail: karabea@lipetsk.ru      http://ikarab.narod.ru
; Используется микроконтроллер PIC16F84A. Частота кварца 4000кГц.
; Объем программы: 46 слов в памяти программ.
;*******************************************************************************
;                          "ШАПКА"  ПРОГРАММЫ
;*******************************************************************************
            LIST       p=16F84A     ; Назначение типа ПИКа: PIC16F84A.
            __CONFIG   03FF5H       ; Установка битов  конфигурации: стандартный
                                    ; XT-генератор, WDT включен, бит защиты не
                                    ; установлен, PWRT включен (1111 0101).
;===============================================================================
; Определение адресов регистров специального назначения.
;===============================================================================
OptionR     equ        01h          ; Регистр Option - банк1
Status      equ        03h          ; Регистр Status
PortB       equ        06h          ; Порт B
TrisB       equ        06h          ; Регистр Tris B - Банк1
IntCon      equ        0Bh          ; Регистр IntCon
;===============================================================================
; Определение названия и адресов регистров общего назначения.
;===============================================================================
Sec         equ        0Ch          ; Счетчик времени полупериода.
SecH        equ        0Dh          ; Старший байт таймера.
SecL        equ        0Eh          ; Младший байт таймера.
;===============================================================================
; Присвоение буквенного обозначения операции направления результата выполнения
; команды в регистр, с содержимым которого производится действие (для удобства
; восприятия текста программы).
;===============================================================================
F           equ        1            ; Результат направить в регистр, с содержимым
                                    ; которого производится действие.
;===============================================================================
; Присвоение биту выбора банка регистра STATUS (пятому) его стандартного названия
; (для удобства восприятия текста программы).
;===============================================================================
RP0         equ        5            ; Присвоение 5-му биту регистра STATUS
                                    ; названия RP0.
;===============================================================================
; Определение точки входа в программу.
;===============================================================================
            org        0            ; Начать выполнение программы
            goto       START        ; с первой команды подпрограммы START.
;*******************************************************************************

;*******************************************************************************
;----------------------- РАБОЧАЯ  ЧАСТЬ  ПРОГРАММЫ ---------------------------
;*******************************************************************************
```

```
; Подготовительные операции.
;-------------------------------------------------------------------------------
START       clrf        IntCon      ; Запрещение всех прерываний.
            clrwdt                  ; Сброс сторожевого таймера WDT.
            bsf         Status,RP0  ; Установка банка 1.

            movlw       .65         ; RB0,RB6 работают на вход,  (.65 = 0100 0001)
            movwf       TrisB       ; остальные - на выход.

            movlw       .143        ; Выключение подтягивающих резисторов порта B.
            movwf       OptionR     ; Предделитель с Кдел.=128 (18мс.*128=2,3сек.)
                                    ; подключен к WDT, остальное - не важно.
                                    ;           (.143 = 1000 1111)
            bcf         Status,RP0  ; Установка банка 0.
;-------------------------------------------------------------------------------
; Запись констант времени работы таймера.
;-------------------------------------------------------------------------------
            movlw       .15         ; Запись в регистр SecH
            movwf       SecH        ; константы .15

            movlw       .255        ; Запись в регистр SecL
            movwf       SecL        ; константы .255
;-------------------------------------------------------------------------------
; Проверка наличия включения на передачу (опрос клавиатуры).
;-------------------------------------------------------------------------------
CYCLE       btfsc       PortB,0     ; Если передача не включена, то переход
                                    ; в ПП START.
            goto        START       ; Если включена - программа исполняется далее.
;-------------------------------------------------------------------------------
; Проверка: режим тонального вызова включен или выключен ?
;-------------------------------------------------------------------------------
            btfss       PortB,6     ; Если режим тонального вызова выключен - уход
                                    ; в PRD.
            goto        PRD         ; Если включен, то программа исполняется далее
;-------------------------------------------------------------------------------
; Формирование времени отрицательного полупериода сигнала тонального вызова.
;-------------------------------------------------------------------------------
            bcf         PortB,2     ; Начало формирования отрицательного
                                    ; полупериода.
            nop                     ; Пустые машинные циклы точной калибровки
            nop                     ; времени отрицательного полупериода
                                    ; (точная доводка).
            nop                     ; -------------------"--------------------
            movlw       .85         ; Запись в регистр Sec
            movwf       Sec         ; константы .85

PAUSE_1     clrwdt                  ; Сброс сторожевого таймера WDT.

            decfsz      Sec,F       ; "Грубый" отсчет интервала времени
            goto        PAUSE_1     ; отрицательного полупериода.
;-------------------------------------------------------------------------------
; Формирование времени положительного полупериода сигнала тонального вызова.
;-------------------------------------------------------------------------------
            bsf         PortB,2     ; Начало формирования положительного
                                    ; полупериода.
            nop                     ; Пустые машинные циклы точной калибровки
            nop                     ; времени положительного полупериода
                                    ; (точная доводка).
            nop                     ; -------------------"--------------------
            movlw       .83         ; Запись в регистр Sec
            movwf       Sec         ; константы .83

PAUSE_2     clrwdt                  ; Сброс сторожевого таймера WDT.

            decfsz      Sec,F       ; "Грубый" отсчет интервала времени
            goto        PAUSE_2     ; положительного полупериода.
```

```
;-------------------------------------------------------------------------
; "Очистка" (декремент) таймера.
;-------------------------------------------------------------------------
            decfsz      SecL,F      ; Декремент содержимого регистра SecL.
            goto        CYCLE       ; Если результат декремента не=0, то переход
                                    ; в ПП CYCLE.
                                    ; Если результат декремента =0, то программа
                                    ; исполняется далее.
            decfsz      SecH,F      ; Декремент содержимого регистра SecH.
            goto        CYCLE       ; Если результат декремента не=0, то переход
                                    ; в ПП CYCLE.
                                    ; Если результат декремента =0, то программа
                                    ; исполняется далее.
            bcf         PortB,2     ; Установить на выходе RB2 ноль.
;-------------------------------------------------------------------------
; Уход рабочей точки программы в "вечное" кольцо и выход из него после
; переключения р/станции с передачи на прием.
;-------------------------------------------------------------------------
PRD         clrwdt                  ; Сброс сторожевого таймера WDT.
            btfss       PortB,0     ; Передача включена?
            goto        PRD         ; Если да, то переход на ПП PRD.
                                    ; Если нет, то программа исполняется далее.
;           Еще одна проверка.

            btfss       PortB,0     ; Передача включена?
            goto        PRD         ; Если да, то переход на ПП PRD.
                                    ; Если нет, то программа исполняется далее.
            goto        START       ; Конец полного цикла программы, переход на
                                    ; новый полный цикл программы.
;=========================================================================
            end                     ; Конец программы.
```

Текст программы cus_2.asm выглядит так:

```
;*************************************************************************
; cus_2.asm                 ВАРИАНТ КОММЕНТАРИЯ № 2
;      Программа разработана для устройства тонального вызова с частотой 1450 Гц.
;*************************************************************************
; Автор: Корабельников Евгений Александрович  г.Липецк,  январь 2005г.
; E-mail: karabea@lipetsk.ru      http://ikarab.narod.ru
; Используется микроконтроллер PIC16F84A. Частота кварца 4000кГц.
; Объем программы: 46 слов в памяти программ.
;*************************************************************************
;                        "ШАПКА"  ПРОГРАММЫ
;*************************************************************************
            LIST        p=16F84A    ; Назначение типа ПИКа: PIC16F84A.
            __CONFIG    03FF5H      ; Установка битов  конфигурации: стандартный
                                    ; XT-генератор, WDT включен, бит защиты не
                                    ; установлен, PWRT включен (1111 0101).
;=========================================================================
; Определение адресов регистров специального назначения.
;=========================================================================
OptionR     equ         01h         ; Регистр Option - банк1
Status      equ         03h         ; Регистр Status
PortB       equ         06h         ; Порт B
TrisB       equ         06h         ; Регистр Tris B - Банк1
IntCon      equ         0Bh         ; Регистр IntCon
;=========================================================================
; Определение названия и адресов регистров общего назначения.
;=========================================================================
Sec         equ         0Ch         ; Счетчик времени полупериода.
SecH        equ         0Dh         ; Старший байт таймера.
SecL        equ         0Eh         ; Младший байт таймера.
;=========================================================================
```

```
; Присвоение буквенного обозначения операции направления результата выполнения
; команды в регистр, с содержимым которого производится действие (для удобства
; восприятия текста программы).
;=============================================================================
F           equ         1           ; Результат направить в регистр, с содержимым
                                    ; которого производится действие.
;=============================================================================
; Присвоение биту выбора банка регистра STATUS (пятому) его стандартного названия
; для удобства восприятия текста программы).
;=============================================================================
RP0         equ         5           ; Присвоение 5-му биту регистра STATUS
                                    ; названия RP0.
;=============================================================================
; Определение точки входа в программу.
;=============================================================================
            org         0           ; Установка нулевого адреса в счетчике
                                    ; команд PC.
            goto        START       ; Безусловный переход на подпрограмму START.
;*****************************************************************************


;*****************************************************************************
;---------------------------- РАБОЧАЯ  ЧАСТЬ  ПРОГРАММЫ ------------------------
;*****************************************************************************
; Подготовительные операции.
;-----------------------------------------------------------------------------
START       clrf        IntCon      ; Сброс в ноль всех битов регистра IntCon.
            clrwdt                  ; Установка начала отсчета сторожевого
                                    ; таймера WDT.
            bsf         Status,RP0  ; Установка 5-го бита регистра Status
                                    ; в единицу.
            movlw       .65         ; Запись в регистр W константы .65
                                    ; (.65 = 0100 0001)
            movwf       TrisB       ; Установка нулевого и 6-го бита регистра
                                    ; TrisB в единицу, а остальных - в ноль.
            movlw       .143        ; Запись в регистр W константы .143
                                    ; (.143 = 1000 1111)
            movwf       OptionR     ; Установка 4,5,6-го битов регистра OptionR
                                    ; в ноль, а остальных - в единицу.
            bcf         Status,RP0  ; Установка 5-го бита регистра Status в ноль.
;-----------------------------------------------------------------------------
; Запись констант времени работы таймера.
;-----------------------------------------------------------------------------
            movlw       .15         ; Запись в регистр W константы .15
            movwf       SecH        ; Копирование константы .15 из регистра W
                                    ; в регистр SecH.
            movlw       .255        ; Запись в регистр W константы .255
            movwf       SecL        ; Копирование константы .255 из регистра W
                                    ; в регистр SecL.
;-----------------------------------------------------------------------------
; Проверка наличия включения на передачу (опрос клавиатуры).
;-----------------------------------------------------------------------------
CYCLE       btfsc       PortB,0     ; Если нулевой бит регистра PortB равен 1, то
                                    ; выполняется следующая команда (goto START),
                                    ; а если равен 0, то следующая команда не
                                    ; выполняется (вместо нее - nop) и следующей
                                    ; активной командой будет btfss PortB,6
            goto        START       ; Безусловный переход на подпрограмму START.
;-----------------------------------------------------------------------------
; Проверка: режим тонального вызова включен или выключен ?
;-----------------------------------------------------------------------------
            btfss       PortB,6     ; Если 6-й бит регистра PortB равен 0, то
                                    ; выполняется следующая команда (goto PRD), а
                                    ; если равен 1, то следующая команда не
                                    ; выполняется (вместо нее - nop) и следующей
                                    ; активной командой будет bcf PortB,2
```

```
            goto        PRD         ; Безусловный переход на подпрограмму PRD.
;-------------------------------------------------------------------------------
; Формирование времени отрицательного полупериода сигнала тонального вызова.
;-------------------------------------------------------------------------------
            bcf         PortB,2     ; Установить 2-й бит регистра PortB в 0.
            nop                     ; Пустые машинные циклы.
            nop                     ; ----------"----------
            nop                     ; ----------"----------

            movlw       .85         ; Запись в регистр W константы .85
            movwf       Sec         ; Копирование константы .85 из регистра W
                                    ; в регистр Sec.
PAUSE_1     clrwdt                  ; Установка начала отсчета сторожевого
                                    ; таймера WDT.
            decfsz      Sec,F       ; Декремент (-1) содержимого регистра Sec с
                                    ; сохранением результата декремента в нем же.
                                    ; Если этот результат не=0, то выполняется
                                    ; следующая команда (goto PAUSE_1), а если =0,
                                    ; то следующая команда не выполняется (вместо
                                    ; нее - nop) и следующей активной командой
                                    ; будет bsf PortB,2
            goto        PAUSE_1     ; Безусловный переход на подпрограмму PAUSE_1.
;-------------------------------------------------------------------------------
; Формирование времени положительного полупериода сигнала тонального вызова.
;-------------------------------------------------------------------------------
            bsf         PortB,2     ; То же самое, что и при формировании
            nop                     ; отрицательного полупериода, только
            nop                     ; 2-й бит регистра PortB
            nop                     ; устанавливается в 1.

            movlw       .83         ; То же самое, что и при формировании
            movwf       Sec         ; отрицательного полупериода, только в регистр
                                    ; Sec записывается константа .83
PAUSE_2     clrwdt                  ; Установка начала отсчета сторожевого
                                    ; таймера WDT.
            decfsz      Sec,F       ; То же самое, что и при формировании
            goto        PAUSE_2     ; отрицательного полупериода, только
                                    ; безусловный переход осуществляется на
                                    ; подпрограмму PAUSE_2.
;-------------------------------------------------------------------------------
; "Очистка" (декремент) таймера.
;-------------------------------------------------------------------------------
            decfsz      SecL,F      ; Декремент (-1) содержимого регистра SecL с
            goto        CYCLE       ; сохранением результата в нем же.
                                    ; Если этот результат не=0, то выполняется
                                    ; следующая команда (goto CYCLE), а если =0,
                                    ; то следующая команда не выполняется (вместо
                                    ; нее - nop) и следующей активной командой
                                    ; будет decfsz SecH,F
            decfsz      SecH,F      ; То же самое, что и при декременте
            goto        CYCLE       ; содержимого регистра SecL, только
                                    ; применительно к регистру SecH.
            bcf         PortB,2     ; Установить 2-й бит регистра PortB в 0.
;-------------------------------------------------------------------------------
; Уход рабочей точки программы в "вечное" кольцо и выход из него после
; переключения р/станции с передачи на прием.
;-------------------------------------------------------------------------------
PRD         clrwdt                  ; Установка начала отсчета сторожевого
                                    ; таймера WDT.
            btfss       PortB,0     ; Если нулевой бит регистра PortB равен 0, то
                                    ; выполняется следующая команда (goto PRD), а
                                    ; если равен 1, то следующая команда не
                                    ; выполняется (вместо нее - nop) и следующей
                                    ; активной командой будет btfss PortB,0
            goto        PRD         ; Безусловный переход на подпрограмму PRD.
```

```
;           Еще одна проверка.

            btfss       PortB,0     ; То же самое, что и при первой проверке,
                                    ; только, при равенстве нулевого бита регистра
                                    ; PortB единице, следующей активной командой
                                    ; будет goto START.
            goto        PRD         ; Безусловный переход на подпрограмму PRD.
            goto        START       ; Безусловный переход на подпрограмму START.

;===============================================================================
            end                     ; Директива конца текста программы.
```

Для того чтобы далее работать с текстами этих программ, создайте проекты **cus_1** и **cus_2**, а затем скопируйте в них файлы **cus_1.asm** и **cus_2.asm** соответственно.
О том, как это делается, рассказано в 5-м разделе.
Можете проассемблировать их и убедиться в отсутствии ошибок.
Итак, перед Вами текст программы **cus** (так я буду называть любую из них), оформленный по всем правилам составления текста программы.
Ранее я упоминал про "правило 12-ти пробелов".
В тексте программы **cus,** это правило использовано на практике.
Давайте разбираться.
Текст программы содержит комментарии, из которых можно многое узнать.
Такого рода комментарии - достаточно большая редкость.
По совокупности причин, разработчики программ редко себя этим утруждают.
Отчасти этим и объясняется то, что при наличии большого количества разнообразных текстов программ, большая их часть так и остается "тайной за семью печатями".
Для опытных программистов, владеющих приемами программно-аппаратного анализа, отсутствие комментариев не является непреодолимой преградой, но с начинающими сложнее: с непривычки, можно "заполучить множество фингалов".
Чтобы уменьшить их количество, давайте потрудимся.
Напоминаю Вам снова и снова: **движение рабочей точки программы, по тексту программы, никогда не останавливается** (за исключением "уходов" в спящий режим SLEEP).
Это означает то, что если необходимо задержать исполнение последующих команд, то рабочую точку программы обязательно нужно "закольцевать" (задержать) в какой-то подпрограмме, образно выражаясь, "поставленной на счетчик".
Выход рабочей точки программы, из этой подпрограммы, возможен либо после завершения, программно организованным счетчиком, своего счета, либо после поступления внешнего, управляющего сигнала.
Любая программа это постоянная "вереница" такого рода колец, и без глубокого понимания смысла и "технологии" этих "закольцовок", в программировании делать нечего.
Программа **cus** не является исключением.
Если Вы поймете ее работу, то найдете ответы на многие вопросы.

"Шапка" программы.

Напоминаю: все что находится правее точек с запятой, **MPLAB** "не видит", и к программе, в буквальном смысле этого слова, это не имеет отношения, так как программа, опять же, в буквальном смысле этого слова, это то, что не заблокировано точками с запятыми.
Вверху "шапки" программы, обычно, помещается общая, пояснительная информация о программе, об устройстве, которое она обслуживает и т.п. ("правило хорошего тона". Не обязательно, но желательно).
Эта информация может быть изложена в произвольной форме.
То же самое относится и к комментариям.
"Активная" часть "шапки" начинается с директив **LIST** и **__CONFIG**.
Первая "подсказывает" **MPLAB**, какой именно файл, из весьма обширной библиотеки файлов, нужно извлечь из ее "недр" для того, чтобы "привязаться" к конкретному типу ПИКа (в нашем случае, к **PIC16F84A**), а при помощи второй директивы устанавливаются значения битов конфигурации.
В **PIC16F84A**, битов конфигурации всего **5** и поэтому, из пяти двоичных разрядов 16-ричного

числа (в данном случае, **03FF5h**), активными (в которых можно что-то менять) являются только **2** последних разряда, а первые три всегда "прописываются" как **03F**.
16-ричному числу **F5h** соответствует бинарное число **1111 0101**.
Сверяемся с распечаткой битов конфигурации.
Получается: переводим тактовый генератор ПИКа в режим стандартного, кварцевого генератора (**XT**), включаем сторожевой таймер **WDT**, разрешаем работу таймера включения питания **PWRT**, отключаем бит защиты памяти программ и **EEPROM** памяти данных.
Что касается **PWRT**, то на него, пока, можно не обращать внимания.
Он не представляет собой нечто совершенно необходимое.
Его можно включить, а можно и выключить.
PWRT, за счет формирования задержки, "нейтрализует" негативные последствия медленного нарастания питающего напряжения.
Практически всегда, эта скорость настолько высока, что "услуг" **PWRT** не требуется.
Далее необходимо определиться с регистрами специального и общего назначения.
Порядок этого "определения" не имеет значения: можно, сначала определиться с регистрами общего, а затем, специального назначения или "перетасовать их как карты".
Так как "бардак", в тексте программы, совсем не нужен, то советую Вам придерживаться того "порядка", который "наведен" в "шапке" программы **cus**.
При "прописке" регистров специального (и общего) назначения, нужно "прописывать" только те регистры, которые задействованы при выполнении программы (к содержимому которых происходят обращения).
"Прописывать" те регистры, которые не задействуются при выполнении программы, можно, но не нужно.
Если возникают сомнения типа: "А вдруг потребуется?", можно "прописать" все (или часть) регистры специального назначения, а потом, когда программа будет отработана, убрать лишние.
Лично я, в подобных обстоятельствах, поступаю именно так.
В программе **cus**, Вы видите только те регистры специального назначения, которые задействуются при выполнении программы. Лишних нет.
"Прописка" происходит при помощи директивы **equ** с указанием адресов регистров специального назначения в области оперативной памяти.
Обратите внимание на то, что адрес регистра **TrisB** (1-й банк) дублирует адрес регистра **PortB** (нулевой банк).
Это то, о чем говорилось ранее (кто забыл, вернитесь назад).
Регистры общего назначения прописываются по адресам области оперативной памяти, не относящимся к регистрам специального назначения.
Можно выбрать любые из этих адресов. Это личное дело программиста.
Регистрам общего назначения нужно присвоить какие-нибудь названия.
Это тоже личное дело программиста.
Ну и соответственно, "прописать" эти названия в "шапке" программы.
Я назвал их **Sec**, **SecH** и **SecL**.
Какие они выполняют функции - понятно из комментариев.
Возникает законный вопрос: "Откуда они взялись и вообще, что это такое"?
А взялись они из программы.
Становится еще непонятней: "А если, например, программа еще не написана, то какие регистры общего назначения прописывать"?
Ответ: никакие.
Вопрос: а откуда взялись **Sec, SecH** и **SecL**?
Ответ: из программы.
Похоже на испорченную пластинку. Даже матом хочется заругаться.
Давайте выходить из этого "замкнутого круга".
С регистрами специального назначения все понятно: у них фиксированные названия и их, изначально, можно "прописать всем скопом", а потом "избавиться" от лишних.
С регистрами общего назначения сложнее.
Пока они не будут "фигурировать" в рабочей части программы, эти регистры находятся как бы в состоянии "бревна/болванки" и представляют собой форму без содержания (толка – никакого).
А вот когда команды программы начинают к ним обращаться и что-то с ними "вытворять" (наполнение формы содержанием), то совсем другое дело.

В этом случае, "прописка" обязательна.
Вашему вниманию предоставлена готовая программа, и по этой причине, вполне понятно, какие именно регистры общего назначения нужно прописывать, а анализируя текст программы, становится понятным и то, зачем они нужны и какие функции выполняют.
А если программист сидит перед "чистым листом бумаги" и только приступает к составлению программы???
В части, касающейся регистров специального и общего назначения, это делается так:
В "шапке" программы, "оптом прописываются" регистры специального назначения.
После окончания составления программы, лишние можно удалить.
Регистры общего назначения не "прописываются", так как пока не ясно, что "прописывать".
Им можно назначить какие-нибудь названия, которые, по ходу составления программы, можно в любой момент изменить (у любого более-менее "набившего руку" программиста, имеется такой "набор" привычных названий), после чего эти регистры "прописываются".
Но это не обязательно. Можно начать и с "пустого места".
Например, по ходу создания текста программы, возникла необходимость в однобайтном счетчике.
Ему тут же придумывается название, например, **ABC**, присваивается адрес (в области оперативной памяти), например, **0Eh** и после этого, в "шапку" программы добавляется строка:

```
ABC     equ     0Eh
```

Всё. Этот регистр "прописан" ("введен в эксплуатацию") и с ним можно работать.
Таким же образом "рождаются и вводятся в эксплуатацию" другие регистры общего назначения, необходимость в "рождении" которых возникает по ходу составления текста программы.
В конечном итоге, к концу процесса написания текста программы, в "шапке" программы "накапливается/оседает" какое-то количество регистров общего назначения, к содержимому которых, по ходу исполнения программы, с помощью команд, происходят обращения.
В тексте программы **cus,** таких регистров "накопилось" **три**.
В других программах, их может быть другое количество. Зависит от замысла.
Пошли дальше.
В "шапке" программы, Вы видите строку `F equ 1`.
Смысл этой строки заключается в следующем.
Откройте список команд и обратите внимание на байт-ориентированные команды, и в частности, на букву **d**, которая является составной частью команды.
В зависимости от значения этого параметра (**0** или **1**), результат исполнения команды может сохраниться либо в аккумуляторе (**W**), либо в том же регистре, с содержимым которого производится действие (**F**).
Для комфортного восприятия текста программы, удобнее, если в качестве параметра **d** будет указываться не **0** или **1**, а **W** или **F** соответственно.
Примечание: с регистром **W** все понятно, это его название воспринимается однозначно, а вот регистр **F** физически не существует.
Буква **F**, в символьном (или в символическом, или в обобщенном) виде, обозначает тот регистр, с содержимым которого производится действие и поэтому, за этой буквой может "скрываться" любой из задействованных в программе регистров.
Например, команду декремента содержимого регистра **ABC**, с сохранением результата декремента в нем же, можно "изобразить" так:

```
decf    ABC,1
```

но гораздо удобнее она будет восприниматься как:

```
decf    ABC,F
```

Для того чтобы перейти к этому более комфортному восприятию места сохранения результата выполнения команды, в "шапке" программы и "прописывается" строка `F equ 1`.
Директива **equ** дает указание **MPLAB**: если в тексте программы имеется буква **F**, то ее нужно "воспринять" как **1**.
Такого же рода манипуляцию можно произвести и с нулем, поставив ему в соответствие букву **W**.
В этом случае, в "шапку" программы нужно добавить `W equ 0`.
При написании программы "с чистого листа", в "шапку" программы, "не мудрствуя лукаво", можно поместить обе этих строки.
По окончании работы над программой, текст программы просматривается на предмет наличия/отсутствия, например, команд, результат выполнения которых помещается в регистр

W.
Если есть хотя бы одна такая команда, то строка **W equ 0** оставляется в "шапке" программы, а если их вообще нет, то удаляется.
То же самое относится и к строке **F equ 1**
В тексте программы **cus**, результаты выполнения команд сохраняются в тех же регистрах, с содержимым которых производятся действия, а регистр **W** не задействуется.
По этой причине строка **F equ 1** оставлена, а строка **W equ 0** удалена (хотя, ее можно и оставить. От этого, в работе программы ничего не изменится).
Идем далее.
Строка **RP0 equ 5**
С учетом сказанного выше, становится понятным, что директива **equ** дает указание **MPLAB**: если в тексте программы имеется комбинация символов **RP0**, то ее нужно воспринять как цифру **5** (в данном случае, **5**-й бит регистра **Status** → переключение банков).
В этом случае, команда переключения в нулевой банк будет выглядеть не как **bcf Status,5**, а как **bcf Status,RP0** (установить в 0 пятый бит регистра **Status**), а команда переключения в первый банк будет выглядеть не как **bsf Status,5** , а как **bsf Status,RP0** (установить в 1 пятый бит регистра **Status**).
Таким же образом, при помощи директивы **equ**, можно "подменить" номера любых битов регистров специального назначения на стандартные названия этих битов (можно даже придумать свои названия, но зачем это нужно?).
Названия битов запоминаются гораздо легче, чем номера битов, которые, к тому же, и дублируются (регистров много, а битов всего 8), поэтому рекомендую Вам оперировать не номерами битов, а их названиями.
К таким "вещам" нужно привыкать с самого начала.
Естественно, что имеет смысл присваивать названия только тем битам, которые задействованы в программе.
В программе **cus**, задействован только один такой бит (**RP0**). Он и "прописан".
Если возникнет такая необходимость, то ходу написания программы, можно заменять номера битов на их названия (естественно, с "пропиской").
А можно изначально прикинуть, какие биты могут потребоваться, "оптом прописать" их, а затем, по окончании работы над текстом программы, убрать те названия битов, обращений к которым нет.
Из сказанного можно сделать вывод: ничего плохого не произойдет, если в "шапке" программы будет "прописано" то, что в программе не используется (ошибки нет).
А вот об обратном такого сказать нельзя: если в рабочей части программы будет использовано то, что в "шапке" программы не "прописано", **MPLAB** обязательно "заругается" и укажет Вам на это.
То есть, после ассемблирования, Вы получите сообщение об ошибке, и **MPLAB** "откажется" создавать HEX-файл.
И еще один вывод, директива **equ** ставит в соответствие одно другому:
- название регистра → его адресу,
- название бита → его номеру,
- название константы (константе тоже можно присвоить название) → ее значению.

Применение этой директивы позволяет составить удобный, для работы и восприятия, текст программы.
И в самом деле, зачем каждый раз, при обращении, например, к регистру, указывать его адрес в области оперативной памяти (можно сделать и так), когда этот регистр можно один раз "прописать", а затем указывать только его название.
Этому названию **MPLAB** автоматически поставит в соответствие адрес того регистра, которому оно присвоено (найдет его в области оперативной памяти).
И последнее, что нужно указать в "шапке": "точку входа" в программу и "точку входа" в подпрограмму прерывания.
О последнем будет отдельный разговор.
Так как в программе **cus** прерывания не используются, то и "точку входа" в подпрограмму прерывания указывать не нужно.
Для того чтобы задать "точку входа" в программу, необходимо, при помощи директивы **org**, указать, с какого адреса (в счетчике команд **PC**) нужно начать исполнение программы (**org 0** - начать с нулевого адреса).
В подавляющем большинстве случаев, указывается нулевой адрес, но можно указать и

другой.
В последнем случае, в памяти программ, все команды программы смещаются вниз на количество ячеек **PC**, указанное в рабочей части директивы **org** (применяется редко и в специфических случаях).
Директива **org** подобна стартовому пистолету: после того, как она "сделает свое дело" можно начинать исполнение рабочей части программы.
Если рабочая часть программы начинается с ПП **START**, то никаких дополнительных "пинков" не нужно: программа начнет исполняться с ПП **START**, как говорится, "своим ходом".
А если ПП **START** находится где-то в "дебрях" текста программы (например, в середине текста), то нужен "пинок", в качестве которого используется команда **goto START**.
В соответствии со сказанным, в программе **cus**, после директивы **org 0**, команда **goto START** не нужна, но в обучающих целях, я ее убирать не стал, так как во многих случаях, эта команда нужна.
Пусть она Вам немного "глаза помозолит". Хуже от этого не будет.
Теперь о синтаксисе.
В "шапке" программы, названия "того-сего" могут быть указаны в удобной для программиста форме.
Например, в "шапке" программы, указано название регистра **IntCon**. Это означает то, что в рабочей части программы, при обращении к этому регистру, обязательно должны буквально соблюдаться правила написания этого слова.
Если Вы укажете его название как **INTCON** или **Intcon**, то после ассемблирования получите сообщение об ошибке.
Если бит "прописан" как **RP0**, то именно в таком виде название этого бита должно "фигурировать" в рабочей части программы.
Если Вы там укажете его название как **rP0** или **Rp0**, то опять же, получите сообщение об ошибке.
Сказанное относится как к названиям рабочих битов, так и к названиям битов флагов.
Примечание: в тексте программы **cus**, биты флагов не используются (операций с флагами нет).
Названия регистров **W** и **F** можно указывать и с большой, и с маленькой буквы.
Например, если в "шапке" указана буква **F**, то в рабочей части программы можно писать **f** или **F**. Ошибки не будет.
То же самое распространяется на адреса и директивы.
Например, можно написать адрес регистра как **0CH, 0cH, 0ch, 0Ch**, а директиву как **equ, Equ, EQu, EQU ...**
В рабочей части программы, названия подпрограмм (и меток) должны буквально совпадать с названиями этих же подпрограмм (меток), которые указываются в командах переходов.
Например, имеется подпрограмма **START**.
Если, при переходе в эту подпрограмму, в команде перехода, указано **Start** (**goto Start**, а не **goto START**), то **MPLAB** выдаст сообщение об ошибке.
Названия подпрограмм (меток) придумывает программист.
Он может придумать, например, название **Start_1**.
Главное, чтобы в командах переходов, это название было бы точно таким же.

Рабочая часть программы.

Рабочая часть программы начинается с подпрограммы **START**, которая, в свою очередь, начинается с подготовительных операций и заканчивается записью, в регистры общего назначения, констант, задающих время работы таймера.
Так как запись этих констант не требует никаких предварительных установок, то эту группу команд можно разместить, например, в начале ПП **START** или "врезать" ее между командами подготовительных операций (то, о чем говорилось ранее). Результат будет одним и тем же.
В данном случае, я разместил эти группы команд так, как указано в тексте программы.
ПП **START** начинается с команды **clrf IntCon**.
Смысл этой команды заключается в следующем: вывод **RB0** порта B задействован как вход.
Одновременно, вывод **PB0** является входом внешнего прерывания (**INT**).
К тому же, вывод **RB6**, по которому организуется прерывание по изменению уровня сигнала на выводах **RB4...RB7**, также задействован как вход.
О прерываниях разговор будет позже, а пока просто примите это к сведению.

Хотя в программе **cus** и не предусмотрен уход в прерывания, специалисты фирмы Microchip рекомендуют, в этом случае (и вообще, в случаях, когда задействованы выводы ПИКа, участвующие в организации прерываний, и когда хотя бы один из них, работает на вход) глобально запретить прерывания.
Для этого достаточно установить в **0** седьмой бит регистра **IntCon** (глобальный запрет прерываний), но чаще всего, сбрасывается в **0** весь байт регистра **IntCon**, что Вы и видите в тексте программы (это, как бы, "двойной запрет" прерываний, см. распечатку регистра **IntCon**).
На первых порах, команду **clrf IntCon** можно всегда вставлять в начало программы, в которой не предусмотрен уход в прерывания (или предусмотрен, но позднее), особо не задумываясь, нужна она или нет.
Хуже от этого не будет, а "перестраховочка" не помешает.
Следующая команда: **clrwdt** (сброс сторожевого таймера **WDT**).
Так как сторожевой таймер **включен** (см. биты конфигурации), то его, по ходу исполнения программы, **нужно периодически сбрасывать**.
Напоминаю, что сторожевой таймер представляет собой "немудреный" RC - одновибратор (ждущий мультивибратор) с перезапуском.
Если его периодически не перезапускать (не сбрасывать), то он закончит формирование импульса, и по заднему его фронту, произойдет сброс программы на начало, то есть, переход на первую команду ПП **START**.
Таким образом, если программа, по каким-то причинам, "зависнет", то **WDT** выведет ее из этого состояния.
В этом и заключается основной смысл его применения.
Так вот, рассматриваемая сейчас команда является первой из нескольких таких же команд, "разбросанных" по тексту программы, что вполне объяснимо, так как **WDT** нужно периодически сбрасывать, чтобы он не закончил формирование импульса.
Если программа работает нормально (не "зависает"), то **WDT** никогда не закончит формирование своего импульса, так как он будет периодически перезапускаться.
Ниже по тексту программы, располагается группа из шести команд.
Это и есть основные команды подготовительных операций.

1. Выводы **RB0** и **RB6** должны работать на вход, а вывод **RB2** должен работать на выход.
Следовательно, речь идет о регистре **TrisB**.
Регистр **TrisA** "со счетов сбрасываем", так как выводы порта А не задействованы, и как именно они будут работать → "по барабану".
2. Необходимо определиться с подтягивающими резисторами, предделителем, активными фронтами и тактовыми сигналами, то есть с содержимым регистра **OptionR**.

Оба этих регистра (**TrisB** и **OptionR**) расположены в 1-м банке (см. область оперативной памяти).
По умолчанию, программа (начиная с первой команды ПП **START**) начинает исполняться в 0-м банке, следовательно, прежде чем работать с содержимым регистров **TrisB** и **OptionR**, необходимо перейти в 1-й банк, что и делается при помощи первой команды из этой группы команд: **bsf Status,RP0** (установка 1-го банка).
Напоминаю, что 5-му биту этого регистра, в "шапке" программы, присвоено название **RP0** (то, о чем говорилось ранее), и поэтому в команде можно указать не номер бита, а более удобное для восприятия текста программы, его название.
Итак, переход в 1-й банк осуществлен.
Далее все очень просто: устанавливаем 0-й и 6-й биты этого регистра в **1** (работа "на вход"), а остальные в **0** ("работа на выход").
Собственно говоря, в ноль нужно установить только 2-й бит, но так как выводы **RB1,3,4,5,7** не задействованы, и по этой причине, не имеет значения, как именно они будут работать, то установим их "всем скопом" в **0**, и дело с концом.
Бинарное число, которое соответствует такой установке (**0100 0001**) переводим в более компактную форму.
В данном случае, число переведено из бинарной системы исчисления в десятичную (см. имеющуюся у Вас таблицу).
Получилось **.65** (вспоминайте: точка перед числом → признак десятичного числа).
А можно оставить его в бинарной форме или перевести в 16-ричную.

Это "дело вкуса" программиста.
Данное число определяется (назначается) программистом, следовательно оно будет называться **константой**.
Запись, в регистр **TrisB**, числа **.65** происходит в 2 приема, так как за один прием записать в него константу нельзя.
Сначала константа **.65** записывается в аккумулятор (регистр **W**)**: movlw .65** (запись константы **.65** в регистр **W**), а затем копируется (после копирования, число **.65** из регистра **W** не удаляется, а остается в нем), из регистра **W,** в регистр **TrisB** (**movwf TrisB**).
Итак, задействованные, в принципиальной схеме, выводы порта B, работают в соответствии с ранее сформулированным требованием.
Переходим к содержимому регистра **OptionR**.
Сразу "выбраковываем" то, что не нужно.
Так как внешние тактовые сигналы в работе устройства не используются (внешние сигналы управления не являются тактовыми) и уходов в прерывания нет, то значения **4-го, 5-го** и **6-го** битов регистра **OptionR** могут быть любыми.
То есть, они не влияют на работу устройства и поэтому установим их, например, в **0** (а можно и в 1 или в любой комбинации).
Теперь работаем с битами, которые на что-то влияют.
Сначала нужно определиться: задействовать или нет предварительный делитель?
Предделитель может быть включен либо перед таймером **TMR0**, либо после сторожевого таймера **WDT**.
Внешних тактовых сигналов у нас нет, значит и незачем включать предделитель перед **TMR0**.
Остается **WDT**.
Давайте поподробнее с ним разберемся.
WDT (сторожевой таймер) сбрасывается командой **clrwdt**.
Строго говоря, вместо слова "сбрасывается", нужно применять слово "перезапускается", но так как по отношению к **WDT**, слово "сбрасывается" является стандартным, то его и буду применять.
Имейте это ввиду.
После инициализации ПИКа, по умолчанию (аппаратно), начинается формирование импульса **WDT**.
Такой аппаратный запуск происходит только один раз (за одно включение питания), а далее, по ходу отработки программы, **WDT** (если он включен) нужно программно перезапускать.
При каждом перезапуске, к сформированному ранее импульсу, проще говоря, добавляется "довесок/хвостик" продолжительностью **18 мс.**, и поэтому формирование импульса продолжается (не завершается).
"Примерно" потому, что стабильность временных характеристик управляемого RC-одновибратора с перезапуском (это основа **WDT**) оставляет желать лучшего.
Разработчики ПИКов предлагают ориентироваться на **18 мс.**
Если **WDT** работает без предделителя, то сбрасывать (перезапускать) его необходимо через промежутки времени менее чем **18 мс**., то есть, достаточно часто.
Это не всегда бывает удобным из-за необходимости применения относительно большого количество команд **clrwdt**.
Если после **WDT** включить предделитель, то максимальный интервал времени перезапуска увеличится. Он будет кратен коэффициенту деления предделителя.
Например, если предделитель с максимальным коэффициентом деления (**128**) подключен к **WDT**, то команды **clrwdt** нужно располагать в тексте программы через промежутки времени меньшие чем **18x128=2304 мс.** (2,3 сек.).
То есть, потребуется меньшее количество команд **clrwdt**, нежели в случае, когда предделитель не подключен к **WDT**.
Если программа "зависнет" (перезапуски **WDT** прекращаются), то в данном случае, примерно через 2,3 сек. после начала "зависания", произойдет аппаратный сброс программы на ее начало (исполнение программа начнется с ПП **START**).
"Не мудрствуя лукаво", применяю это на практике (можно задать и меньший Кдел. предделителя).
Итак, определились: **подключаем предделитель, с коэффициентом деления 128, к WDT.**
Для этого необходимо установить в **1 первые 4 бита** (начиная с младшего) регистра **OptionR**.
Остался один **7**-й бит: **включение/выключение подтягивающих резисторов порта B.**

Если подтягивающие резисторы порта B программно включены, то они автоматически (аппаратно) отключаются от тех выводов порта B, которые работают "на выход" и остаются подключенными к тем выводам порта B, которые работают "на вход".
Таким образом, если подтягивающие резисторы программно подключены к выводам порта B, то влияние этих резисторов на внешние устройства, подключенные к выводам порта B, работающим "на выход", можно не учитывать.
А вот для внешних устройств, подключенных к выводам порта B, работающим "на вход", это важно.
Если оконечные каскады этих внешних устройств имеют "свою" нагрузку (например, в цепи коллектора или в цепи стока), то подключение подтягивающих резисторов может изменить их режим работы по постоянному току.
В большинстве случаев, такие изменения незначительны (номинал резистора внутренней "подтяжки" – несколько килоом), но в некоторых случаях, могут быть "бяки".
Если эти "бяки" возможны, то подтягивающие резисторы порта B включать не нужно.
А вот если оконечные каскады внешних устройств, подключенных к выводам порта B, работающим "на вход", имеют открытый выход (каскады с открытым коллектором или открытым стоком. "Своя" нагрузка отсутствует), то "подтяжку" можно/нужно включить.
В этом случае, внутренние, подтягивающие резисторы порта B можно/нужно использовать в качестве нагрузок оконечных каскадов внешних устройств, что очень удобно, так как в этом случае, внешняя "подтяжка" не нужна (принципиальная схема устройства упрощается).
То же самое относится и к внешним, механическим, коммутационным устройствам типа кнопок/тумблеров.
В нашем случае, подтягивающие резисторы порта B включать не нужно, так как выходы внешних устройств, подключенных к выводам **RB0** и **RB6**, имеют "свои" нагрузки.
Следовательно, **7**-й бит регистра **OptionR** нужно установить в **1**.
То есть, в регистр OptionR нужно записать константу **.143** (**1000 1111**).
Можно использовать также и число **8Fh** (см. таблицу перевода).
Так как это число является константой, то запись производится через аккумулятор (**W**).
Получается то, что Вы видите в тексте программы: **подтягивающие резисторы порта B выключены, и предделитель, с коэффициентом деления 128, включен после WDT.**
В 1-м банке "все дела сделаны" и делать в нем больше нечего.
Теперь нужно перейти в основной (нулевой) банк: **bcf Status,RP0** (выбор нулевого банка).
Идем дальше.
Теперь необходимо определиться с числовыми значениями тех констант, которые определяют величину интервала времени "выхода" сигнала тонального вызова в эфир.
Реально, при составлении программы "с чистого листа", ПП **START** так и заканчивается командой **bcf Status,RP0**.
И это естественно, так как пока неизвестно, что из себя будет представлять таймер.
Только после того, как "конструкция" и принцип работы таймера будут определены и задействованные в нем регистры общего назначения, будут "прописаны", можно рассуждать о записи, в эти регистры, каких-то констант.
Забегая вперед, скажу, что таймер "собран" на регистрах общего назначения **SecH** и **SecL**.
После их "прописки", а также и "конструирования", на их основе, таймера, группа команд записи констант (4 команды) просто-напросто "врезается" в концовку ПП **START** (см. текст программы).
Давайте сделаем так: не будем "ставить телегу впереди лошади", а предположим, что программа пишется "с чистого листа" и принцип работы таймера пока не известен.
В этом случае, сразу же после команды **bcf Status,RP0**, минуя команды операций с константами таймера и рассчитывая на то, что впоследствии, эти команды будут "вставлены" в концовку ПП **START** (так и происходит при реальном составлении текста программы), переходим к составлению подпрограммы **CYCLE**,.
Итак, в левом столбце текста программы, "настукиваем" название подпрограммы (**CYCLE**).
Смотрим в блок - схему программы.
Сначала необходимо проверить, управляющий сигнал включения на передачу есть, или его нет?
Ранее мы определились, что сигналу включения на передачу соответствует **0**, а выключения → **1**. Этот сигнал присутствует на выводе **RB0**, который настроен на работу "на вход".
Следовательно, нужно опросить состояние этого вывода, и в зависимости от результата

этого опроса, "уйти" в один из двух возможных сценариев работы программы.
Вот Вам и типичный пример необходимости применения команды **ветвления**.
Теперь нужно выбрать, какую из команд ветвления применить?
Для этого нужно предельно четко и ясно сформулировать алгоритм производимой логической операции.
Он формулируется следующим образом:

1. Если нулевой бит регистра **PortB** (он соответствует выводу **RB0**) равен **1** (передача выключена), то тональный сигнал формироваться не должен, и рабочая точка программы должна где-то "закольцеваться" (напоминаю, что выполнение программы не должно останавливаться), до момента появления, на выводе **RB0**, нулевого уровня.
2. Если нулевой бит регистра **PortB** равен **0** (передача включена), то должно начаться формирование тонального сигнала (программа должна исполняться далее).

Эта формулировка достаточно конкретна, но не совсем.
Лирическое отступление: любой микроконтроллер, также как и любая микро-ЭВМ, это всего-лишь "бездушная железяка".
И эта "железяка абсолютно не терпит" неопределенности.
Для того чтобы она заработала, необходимо избавиться от любой неопределенности.
В конечном итоге, задача любого программиста сводится к "борьбе" с неопределенностями и к умению четко формулировать алгоритмы работы всех составных частей программы.
Это присутствует всегда, но особенно ярко это проявляется при работе с командами ветвления.
Итак, устраняем неопределенность, которая заключается в том, что пока, не понятно, каким образом и куда именно необходимо перейти рабочей точке программы, для того чтобы там "закольцеваться"?
Если речь идет о том, что нужно куда-то перейти, то сама собой возникает мысль о том, что этот переход можно осуществить, используя **команду перехода**.
Вопрос: какую именно команду перехода использовать?
Анализируем.
"Закольцовывать" рабочую точку программы в командах, расположенных ниже (по тексту программы) нельзя, так как ниже будут располагаться команды, формирующие сигнал тонального вызова, а в соответствии со сформулированным выше алгоритмом, при наличии, на выводе **RB0**, единицы, сигнал тонального вызова формироваться не должен.
Остается только одно "место" → подпрограмма **START**.
Следовательно, нужно перейти в нее.
В этом случае, рабочая точка программы будет "крутиться по кольцу", "границы" которого: от первой команды ПП **START** и до команды перехода в ПП **START**.
Теперь остается только выяснить, какой из двух видов переходов применить (безусловный или условный)?
Условный переход, то есть, переход с использованием стека, применять нет смысла хотя бы потому, что для возврата по стеку требуется лишняя команда (команда возврата), да и вообще, смысла задействовать, для выполнения элементарной "закольцовки", стек, нет.
Вполне достаточно команды безусловного перехода **goto**.
В данном случае (на выводе **RB0** → **1**), команда **goto**, на каждом "витке", просто будет "отфутболивать" рабочую точку программы на первую команду ПП **START**.
До тех пор, пока, на выводе **RB0**, **1** не сменится на **0**.
Итак, устраняем последнюю неопределенность: при наличии, на выводе **RB0**, единицы, должен быть осуществлен безусловный переход в ПП **START**.
В конечном виде, алгоритм проверки формулируется так:

- ***если бит №0 регистра PortB*** (вывод **RB0**) ***равен 1*** (передача выключена)***, то должен быть осуществлен безусловный переход в ПП START,***
- ***а если бит №0 регистра PortB равен 0*** (передача включена)***, то программа должна исполняться далее.***

Всё... Полная определенность.
Теперь нужно просто подобрать нужную команду ветвления.
Естественно, что эта команда должна быть бит-ориентированной (происходит обращение к отдельному биту).
Смотрим в список команд.
Таких команд две: **btfsc** и **btfss**.
Команда **btfss** не подходит, так как сразу же после команды ветвления, необходимо будет

устанавливать две команды переходов, устанавливать метку и вообще, неэффективно "ломать голову по поводу выхода из вилки". Зачем нужен этот "геморрой"?
А вот **btfsc** - в самый раз.
В тексте программы **cus**, посмотрите на первые две команды ПП **CYCLE**.
Прочитайте комментарии. Мне к ним добавить просто нечего.
Не правда ли, как просто.
Единственное, что можно добавить: словосочетание "программа исполняется далее" означает то, что следующей будет исполняться команда **btfss PortB,6**
Если мыслить "глобально", то есть, не "привязываясь" к данной программе, а имея ввиду программы вообще, то на месте этой команды может оказаться любая другая команда.
Прошу обратить внимание на одну особенность команд ветвления, связанную с переходами.
Суть: любая команда ветвления предполагает наличие двух сценариев дальнейшей работы программы.
Если одним из этих сценариев является сценарий типа "программа исполняется далее", то вторым сценарием, в большинстве случаев (но не всегда), является сценарий типа "переход куда-то" (в подпрограмму, на метку), причем, команда, на которую переходит рабочая точка программы по сценарию "программа исполняется далее", должна быть второй снизу (по тексту программы), от команды ветвления, так как только в этом случае программа может "исполняться далее".
В случае наличия и на первом, и на втором "месте" (снизу от команды ветвления) команд переходов, сценарий "программа исполняется далее" не может быть выполнен.
В этом случае, осуществляется либо первый, либо второй переход.
Таким образом, сценарий типа "программа исполняется далее" связан с исполнением второй, после команды ветвления, команды.
Идем дальше.
Смотрим в блок-схему программы.
Следующая операция - проверка включения или выключения режима тонального вызова.
Ход рассуждений точно такой же, как и выше.
Разница заключается в следующем.
Работа происходит с битом **№ 6** регистра **PortB**.
Уровни управляющего сигнала "привязаны" к другому органу управления.
Переход, с целью "закольцовки" рабочей точки программы в "вечном кольце", осуществляется в подпрограмму **PRD**.
Для реализации алгоритма проверки, использована команда **btfss**.
А теперь - потренируйтесь.
Постарайтесь самостоятельно понять, "где здесь собака порылась" и почему нужно организовывать эту проверку в том виде, в котором она организована в тексте программы (почему не используется команда **btfsc**?).
Указанные выше, 2 проверки, идущие подряд одна за другой, выполняют роль своеобразного "сита".
Результат их совместной работы таков, что для того чтобы началось формирование сигнала тонального вызова, необходимо одновременное выполнение 2-х условий:
- режим тонального вызова должен быть включен,
- и р/станция должна быть включена на передачу.

Подобного рода "сита", при составлении программ, применяются сплошь и рядом.
Если нужно обеспечить одновременное выполнение, например, 3-х или 4-х условий, то применяются 3 или 4 проверки.
В большинстве случаев, такие проверки "идут друг за другом", но они могут быть и разнесены.
О подпрограмме **PRD**.
Загляните в блок-схему программы.
ПП **PRD** нужна для того, чтобы, после 3-хсекундной "выдачи" в эфир сигнала тонального вызова, рабочая точка программы либо "закольцевалась" в этой подпрограмме (если передача включена), либо "ушла" на новый "виток" полного цикла программы (если передача выключена).
Если текст программы составляется "с чистого листа" и программист находится на теперешней стадии ее составления, то при проверке включения или выключения режима тонального вызова, в команде безусловного перехода **goto,** просто указывается название подпрограммы (**PRD**), а саму эту подпрограмму можно составить позднее.

То есть, тогда, когда "до нее дойдет дело" (что и будет сделано в дальнейшем).
Таким образом, работу по составлению рабочей части программы целесообразно начать с ПП **START,** а в дальнейшем, отрабатывать рабочую часть программы последовательно.
То есть, по ходу исполнения программы.
При этом, можно пропускать команды, групп команд и даже целые подпрограммы, работающие с содержимым регистров, которые еще не "введены в бой" (но будут "введены" в будущем), в том числе и с переходами в подпрограммы, которые еще не созданы (но будут созданы в будущем).
Можно пропускать даже несколько подпрограмм, с расчетом на их создание/проработку в будущем (приходит с опытом).
В любом случае, нужно быть уверенным в том, что "стратегических ляпов" не допущено (блок-схема программы капитально продумана).

8. Пример создания программы (продолжение)

Итак, мы остановились на том, что, в ходе описанных выше проверок, состояния выводов **RB0** и **RB6** анализируются, и в зависимости от соотношения уровней сигналов на этих выводах, рабочая точка программы либо "закольцовывается" в ПП **START**, либо "закольцовывается" в ПП **PRD** (в обеих случаях → "уход в вечное кольцо", с выходом из него по внешнему воздействию), либо она "движется далее" по тексту программы (сценарий типа "программа исполняется далее").
Первый случай был описан выше. Это самая простая "закольцовка".
Во втором случае, при условии, что режим тонального вызова выключен, а передача включена, рабочая точка программы "уходит в вечное кольцо" ПП **PRD**.
После выключения с передачи, рабочая точка программы должна "прыгнуть" в ПП **START**, в которой должна "уйти в еще одно вечное кольцо", выход из которого происходит в момент переключения с приема на передачу.
В третьем случае, рабочая точка программы должна продолжить свое "движение" вниз по тексту программы.
Смотрим в блок - схему программы.
Теперь необходимо сформировать один период сигнала тонального вызова, соответствующий частоте 1450 Гц.
То есть, сначала нужно сформировать первую половину периода (полупериод), а затем, вторую (форма сигнала – "меандр").
В конечном итоге, **задача сводится к изменениям уровней сигнала на выводе RB2 и фиксации (задержке) этих уровней на время полупериода.**
Вывод **RB2** работает "на выход" (см. подготовительные операции), а это означает то, что к этому выводу подключен выход защелки, которая управляется вторым битом регистра **PortB**.
Таким образом, устанавливая бит **№2** регистра **PortB** в **0** или **1** и задерживая момент следующей смены состояний этого бита, можно сформировать нужный период.
Зависимость простая: если бит **№2** регистра **PortB** установить в **0**, то на выходе соответствующей защелки, а следовательно и на выводе **RB2,** установится **0**, а если **1**, то установится **1**.
Напоминаю, что защелка это триггер, и если он установлен управляющим импульсом (командой, которая его формирует) в какое-то одно из двух своих состояний, то он будет находиться в этом состоянии вплоть до поступления следующего управляющего импульса (формируется другой командой).
Таким образом, единожды сформировав, на выходе защелки, например, нулевой уровень, в дальнейшем (в интервале времени его фиксации), подтверждать, в каждом машинном цикле, этот уровень, не нужно. Вплоть до смены нулевого уровня на единичный.
Из этого следует то, что после установки, на выводе **RB2**, например, того же нулевого уровня, рабочая точка программы может "скакать" по каким-то другим командам, производя при этом действия, не связанные с управлением защелками порта В.
Следовательно, после смены уровней на выводе **RB2**, можно, на определенное время, "закольцевать" рабочую точку программы в подпрограммах задержки, и за счет этого, сформировать интервалы времени полупериодов. А значит и период.
Этим сейчас и займемся.
Сначала нужно определиться, с какого уровня начинать формирование полупериода: с нулевого или с единичного?
В данном случае, ничего синхронизировать не нужно.
Значит, формирование периода можно начать с любого уровня.
Начнем, например, с нулевого.
Для того чтобы начать формирование нулевого уровня на выводе **RB2**, необходимо установить в ноль бит **№2** регистра **PortB**.
Делаем это при помощи бит-ориентированной команды **bcf** (**bcf PortB,2**).
Далее Вы видите три **NOP**а.
При составлении программы с "чистого листа", изначально, их нет.
Это "элементы" точной установки ("подгонки") интервала времени отрицательного полупериода под заданное значение.
Они "врезаются" в текст программы после "прогонки" подпрограммы задержки через секундомер **MPLAB**а (об этом, позднее).
Теперь нужно "закольцевать" рабочую точку программы в подпрограмме задержки, которая

ранее названа **PAUSE_1**.
Ее "конструкция" должна быть Вам понятна.
Эта подпрограмма задержки построена "по образу и подобию", рассмотренной ранее, подпрограммы задержки программы **Multi.asm**.
Следовательно, необходимо, в качестве счетчика, назначить регистр (регистры) общего назначения.
Количество этих регистров зависит от величины времени задержки.
Чем большее время задержки нужно обеспечить, тем большее количество регистров требуется.
Расчет их количества и "грубое" определение величины константы/констант:
Период сигнала с частотой 1450 Гц. = **689,6 мкс**., а полупериод → **344,8 мкс**.
Так как в данном случае, время отработки одного машинного цикла равно 1 мкс., то округляем величину интервала времени полупериода до **345 мкс**.
Именно такую величину задержки и нужно сформировать.
Полный цикл рассматриваемой подпрограммы задержки составляет 3 м.ц. (3 мкс.), плюс 2 м.ц. (2 мкс.) на последнем "витке".
Если применить конструкцию ПП задержки программы **Multi.asm**, то одним регистром можно обеспечить задержку до 256x3-1=767 мкс.
Полупериод равен 345 мкс., следовательно, **достаточно одного регистра общего назначения**.
Если речь идет о задержках, то уместно вспомнить о том, что не мешало бы сбросить **WDT** (вспомните, по ходу выполнения программы, его нужно периодически сбрасывать).
Команду **clrwdt** можно поставить следующей, после команды **bcf PortB,2** с сохранением 3-хмикросекундного, полного цикла ПП задержки, а можно и "врезать" команду **clrwdt** в полный цикл ПП задержки, увеличив тем самым полный цикл ПП задержки с 3 мкс. до 4 мкс. (команда **clrwdt** выполняется за 1 м.ц.).
В обоих случаях, ПП задержки будет работать, но естественно, что при формировании ими одинакового времени задержки, величины их констант не будут равны.
По большому счету, выгоднее "врЕзать" команду **clrwdt** в цикл ПП задержки, чем размещать ее после команды **bcf PortB,2**.
В этом случае, сброс **WDT** будет происходить чаще (1 раз за 4 мкс., вместо 1 раза за 690 мкс.).
В тексте программы **cus**, Вы видите именно такой вариант построения ПП задержки (с "врезкой"), который я применил не столько по причине необходимости столь частого сброса **WDT** (такой необходимости нет), сколько в обучающих целях.
Дело в том, что "врезая" в нее другие команды ("по образу и подобию" указанной выше "врезки"), время отработки цикла ПП задержки можно сделать гораздо бОльшим, чем 3 м.ц.
В этом случае, и производится полезное действие, и увеличивается время отработки цикла подпрограммы задержки.
А ведь вместо одной команды можно "врезать" и несколько.
С прибавлением каждой такой команды, время отработки цикла ПП задержки будет увеличиваться.
И чем "массивнее врезка", тем бОльшим будет это увеличение.
Если в интервале времени отработки ПП задержки, не нужно производить никаких действий (нужно просто увеличить время отработки ПП задержки), то "врезается" **NOP** (**NOP**ы).
Таким же образом, в подпрограмму задержки, можно "врезать" группу команд, производящих действия, подпрограмму (в том числе и циклическую), группу подпрограмм, а также и все это в комплексе и в любой комбинации (оценИте открывшуюся перспективу …).
В качестве примера, можно привести любую программу частотомера.
В любой из них, за время формирования измерительного интервала времени, выполняется несколько подпрограмм иного, чем задержка, функционального предназначения.
Всё, что касается такого рода "врезок", очень важно, и на это обязательно нужно обратить самое пристальное внимание.
Для того чтобы "закрыть" эту тему, приведу практический пример:
Предположим, что используется классическая конструкция ПП задержки (как в программе **Multi.asm**), обеспечивающая максимальное время задержки 767 мкс., а нужно сформировать задержку в 1000 мкс.
Можно задействовать второй регистр общего назначения, а можно обойтись и одним, "встроив" в ПП задержки всего один **NOP**.

В этом случае, полный цикл ПП задержки увеличивается с 3-х до 4-х м.ц., и при помощи такой ПП задержки можно сформировать задержки до 256х4-1=1023 м.ц. (1023 мкс.)
Задержка в 1000 мкс. попадает в этот интервал, следовательно, можно обойтись одним регистром общего назначения.
Если в ПП задержки "врезается" группа команд, то с целью обеспечения "правильной закольцовки", название ПП задержки должно приходиться на первую команду этой группы команд, а после последней команды этой группы, должна следовать команда ветвления.
Двигаемся дальше.
Ранее было выяснено, что для того чтобы обеспечить задержку в **345 мкс**., достаточно одного регистра общего назначения (даже без "врезки").
Если, при 4-х машинных циклах полного цикла ПП PAUSE_1 (с "врезкой" clrwdt), обеспечивается максимальная задержка в 1023 мкс. (что соответствует максимальному значению константы .255), то значению времени задержки 345 мкс. будет соответствовать значение константы 345х255:1023=85,99...
Округляем до 86-ти.
При написании программы "с чистого листа", значение этой константы (**.86**) и нужно "прописать" в тексте программы.
В тексте программы **cus**, Вы видите другое значение константы (**.85**) и три **NOP**а.
Такого рода числовая коррекция произойдет позднее, при отладке временных характеристик программы в симуляторе (я расскажу об этом подробно), а пока используем результат "грубой прикидки" (**.86**).
Теперь остается только придумать название этого регистра (назовем его **Sec**) и "прописать" его в "шапке" программы ("прописываем"его по адресу **0Ch**, но можно назначить и другой).
Вот вам и ответ на вопрос: "Откуда, в "шапке" программы, взялся регистр **Sec** и каков механизм его возникновения".
После этого, константа **.86**, обычным образом (за 2 приема, через регистр **W**), записывается в "новорожденный" регистр **Sec**.
Эту запись нужно произвести до первой команды подпрограммы (**PAUSE_1** или **PAUSE_2**), в которой используется этот регистр.
Далее располагаются 3 команды ПП **PAUSE_1**, комментарии к которым Вы найдете в тексте программы и к которым мне добавить нечего.
Идем дальше.
Теперь необходимо сформировать **положительный полупериод**.
По своей конструкции, ПП **PAUSE_2** такая же, как и ПП **PAUSE_1**, только, перед ее началом, в бит **№2** регистра **PortB**, записывается **1,** и безусловный переход осуществляется на начало ПП **PAUSE_2**.
Регистр **Sec** уже имеется в наличии, так что ничего "назначать и прописывать" не нужно.
На момент перехода рабочей точки программы на команду **bsf PortB,2** , в регистре **Sec** будет "лежать" ноль (конечный результат декремента содержимого регистра **Sec** в ПП **PAUSE_1**).
Поэтому, перед "влётом" в ПП **PAUSE_2**, в регистр **Sec**, нужно записать "новую" константу.
Она будет определять продолжительность положительного полупериода.
При написании программы "с чистого листа", в качестве этой константы, с расчетом на осуществление дальнейшей коррекции числового значения константы, можно использовать всё то же число **.86**.
Изначально, **NOP**ов также нет. Все это оставляется "на потом".
Почему, в тексте программы **cus**, Вы видите число **.83**, узнаете позже.
Примечание: классическая подпрограмма задержки (такая, как в программе **Multi.asm**), и она же, но с "врезкой" (такая, как в программе **cus**), по своей сути, есть вычитающий счетчик импульсов, то есть, абсолютно необходимое, в цифровой технике, устройство.
Восприятию этого факта мешает то, что импульсов, которые нужно считать, как-будто бы и нет, а есть команды, что на первый взгляд, не одно и то же.
Но на самом деле, импульсы есть, и считаются именно они, так как результатом исполнения команды (например, такой как **decfsz**) является активный перепад (строб), аппаратно формируемый внутри микроконтроллера (его прохождение нельзя проконтролировать с помощью приборов), который и уменьшает (или увеличивает, если применяется команда **incfsz**) на единицу содержимое регистра-счетчика (в данном случае, **Sec**).
Также следует иметь ввиду, что в части касающейся м/контроллеров, счетчик реализуется не одними только аппаратными средствами (например, как в 555ИЕ2), но и программными средствами (в комплексе).

И в самом деле, для того чтобы создать счетчик, необходимо не только задействовать (назначить), в качестве счетчика, регистр/регистры общего назначения (все регистры области оперативной памяти реализованы аппаратно), но и "встроить" его/их в циклическую ПП задержки, в состав которой должны входить байт-ориентированные команды ветвления **incfsz** и/или **decfsz** (они управляют регистром/регистрами).
В первом случае, получается суммирующий, а во втором случае, вычитающий счетчик.
Практический вывод из этого следующий.
Для создания однобайтного счетчика, необходимо назначить, в качестве счетчика, регистр общего назначения, определить его начальную установку, предварительно (до входа в цикл счета) записав в этот регистр константу, и программно организовать циклическую подпрограмму задержки, с использованием команд incfsz (суммирующий счетчик) ***или decfsz*** (вычитающий счетчик).
Если речь идет о многобайтном счетчике, то все то же самое, только во множественном числе.
Примите к сведению: если счетчик, например, двухбайтный (используются 2 регистра общего назначения), то он не обязательно должен быть только суммирующим или только вычитающим. Он может быть еще и комбинированным: один его разряд может работать на суммирование, а другой, на вычитание (или наоборот). Такие счетчики используются редко.
Пример реализации двухразрядного, вычитающего счетчика будет рассмотрен ниже.
ПП задержки не должна отрабатываться все время.
Если такое произойдет, то рабочая точка программы просто "зависнет" в этой подпрограмме.
Счетчик должен считать, от предварительно записанного в него числа (так называемая **предустановка**), до момента установки в нем нуля ("очищение"), после чего рабочая точка программы должна выйти из ПП задержки по сценарию "программа исполняется далее".
Даже в случае "ухода" рабочей точки программы в "вечное кольцо" (специфическая разновидность счетчика без предустановки. "Ловушка рабочей точки"), она все-равно рано или поздно из него выходит. По внешнему воздействию (например, после нажатия кнопки).
Понятия "счетчик" и "задержка" - два "сиамских близнеца" и их нельзя отделить друг от друга.
И в самом деле, счетчик, при условии, что он когда-то "остановится", всегда обеспечивает какую-то задержку, а задержка какого-то процесса на время, кратное машинному циклу, предполагает их подсчет, то есть, применение счетчика.
Таким образом, ПП **PAUSE_1** или **PAUSE_2** можно описать так: ***циклическая подпрограмма задержки с "врезкой" из одной команды, на основе однобайтного, вычитающего счетчика, с предустановкой и выходом из полного цикла подпрограммы после его очистки*** (после окончания счета).
Если "привязаться к понятию "закольцовка", то в части касающейся стандартных ПП задержек, можно сказать так: ***"закольцовка" рабочей точки программы в ПП задержки, создает задержку выполнения следующей, после ПП задержки, команды. Суть "закольцовки" - многократная отработка цикла ПП задержки, вплоть до очищения регистра общего назначения, выполняющего функцию счетчика.***
В этом случае, речь идет о калиброванном времени задержки.
Пример: ПП **PAUSE_1** и **PAUSE_2**.
Если речь идет о "вечном кольце", то такие "закольцовки" есть в ПП **START** и **PRD**.
Если кто-то из Вас не до конца понял правила функционирования этих "механизмов", то вернитесь назад.
В идеале (пусть не сейчас, а в будущем), у Вас должно сложиться свое индивидуальное, образное восприятие подобного рода процессов, которое очень помогает при составлении текстов программ.
Обращаю Ваше внимание на следующее: такого рода общие рассуждения следует рассматривать не как мешающее отвлечение от "разборок" с текстом программы, а как насущную необходимость, связанную с капитальным осознанием смысла того, о чем идет речь.
Без этого осознания, работа конструктора напоминает езду на машине в условиях густого тумана.
Прошу отнестись к этим общим рассуждениям/выводам со всей серьезностью.
Если Вы прислушаетесь к этому доброму совету, то в дальнейшем, у Вас не будет проблем с пониманием текстов следующих программ, так как при описании их работы, я буду исходить из того, что Вы усвоили предыдущую информацию.

А вот теперь, со спокойной совестью, можно перейти к дальнейшим "разборкам" с программой **cus**.
Мы остановились на том, что сформировали оба полупериода (то есть, один период) тонального сигнала вызова.
Естественно, что этого маловато.
Нужно последовательно сформировать несколько тысяч таких периодов, и причем так, чтобы обеспечить заданную скважность ("меандр". Полупериоды равны друг другу).
Так как интервал времени "выдачи" сигнала тонального вызова в эфир вовсе не обязательно в точности делать равным 3 секундам, то задачу можно сформулировать так: сигнал тонального вызова должен "выдаваться" в эфир в течение приблизительно 3-х секунд.
Не трудно догадаться, что после окончания формирования одного периода, нужно сразу же начать формирование следующего периода, и т.д.
До тех пор, пока не сформируется трехсекундный интервал времени "выдачи" сигнала тонального вызова в эфир.
Теперь переходим к устранению всех неопределенностей, а иначе ПИК "не поймет, что он должен сделать и будет с Вами конфликтовать".
Предположим, что мы работаем "с чистого листа".
На данный момент составления текста программы, известно, что при переходе с приема на передачу (на выводе **RB0** **1** меняется на **0**), должен быть запущен некий счетчик времени (таймер), который отмеряет приблизительно 3 секунды.
По окончании формирования этого интервала времени, должно быть выполнено следующее:

- **если р/станция включена на передачу**, то рабочая точка программы должна "уйти в вечное кольцо" подпрограммы **PRD** (ее еще нет. Не "родилась" еще. Мы просто дали ей название, а что в ней → "тайна покрытая мраком") и выйти из него при переключении с передачи на прием (переход на "новый", полный цикл программы),
- **если р/станция включена на прием**, то рабочая точка программы сразу же должна уйти в "вечное кольцо" подпрограммы **START** и выйти из него при переключении с приема на передачу (см. блок-схему программы).

Детализируем.
Так как сначала нужно сформировать трехсекундный интервал времени "выдачи" сигнала тонального вызова в эфир, а только после этого производить "уход в вечное кольцо", то группа команд, производящих операции с таймером, должна быть расположена в тексте программы сразу же после последней команды ПП **PAUSE_2**, а группа команд ПП **PRD** должна следовать сразу же после последней команды группы команд, производящих операции с таймером.
Теперь порядок следования ясен и можно перейти к конструированию таймера.
Для тех, кто хорошо усвоил предыдущую информацию, должно быть понятно, что если речь идет о формировании трехсекундного интервала времени, то необходимо "родить" счетчик, который подсчитывал бы количество периодов (это и есть то, что я называю таймером).
Так как необходимо сформировать фиксированный интервал времени и после этого "уйти" в сценарий "программа исполняется далее" (перейти в ПП **PRD**), то очевидно, что счетчик можно построить "по образу и подобию" рассмотренного выше счетчика (см. ПП **PAUSE_1** и **PAUSE_2**).
В этом случае, все очень просто: все 3 команды подпрограммы, например, **PAUSE_1** (вместе с названием ПП) просто вставляются (копируются) в текст программы сразу же после команды **goto PAUSE_2**.
Теперь нужно "навести порядок".
Название ПП (**PAUSE_1**) необходимо либо заменить (текст программы не должен содержать две ПП с одинаковым названием), либо удалить из текста программы.
Кроме того, нужно определиться, на какую ПП (или метку) необходимо осуществить безусловный переход (**goto**)?
Считать необходимо количество периодов, следовательно, для организации "закольцовки" счетчика, формально, нужно перейти на ту команду, с которой начинается формирование периода. То есть, на команду **bcf PortB,2**
Посмотрите в текст программы **cus**.
Эта команда не является первой командой подпрограммы, и она ничем не помечена (метка не установлена).
Для того чтобы на нее перейти, необходимо придумать какое-нибудь название для метки и "пометить" ей данную команду.

Например, **Metka_1 bcf PortB,2**
После этого, можно осуществить безусловный переход на метку (**goto Metka_1**).
Это конечно сделать можно, но есть более выгодный вариант - переход в ПП **CYCLE**.
В этом случае, задействуются команды обеих проверок, которые рассматривались выше.
Этот вариант выгоден тем, что если выключение с передачи происходит во время формирования сигнала тонального вызова (во время 3-хсекундного интервала времени), то в момент перехода с передачи на прием, формирование сигнала тонального вызова тут же прекращается.
За счет "закольцовки" рабочей точки программы (уход в "вечное кольцо") в ПП **START** ("зона" кольца: от 1-й команды ПП **START** до команды **goto START**).
В этом случае, метку выставлять не нужно.
В случае перехода на команду **bcf PortB,2**, "помеченную" меткой, трехсекундный интервал сформируется полностью, и только после этого рабочая точка программы "уйдет в вечное кольцо" ПП **START**.
С практической точки зрения, вариант с переходом на метку неудобен тем, что в некоторых случаях, пьезоэлектрический излучатель будет выдавать тональный сигнал тогда, когда передатчик не работает (работа на прием), что не совсем удобно для пользователя.
Итак, осуществляем безусловный переход в ПП **CYCLE** (заменяем команду **goto PAUSE_1** на команду **goto CYCLE**).
Название подпрограммы (**PAUSE_1**), по причине ненужности (безусловный переход осуществляется в ПП **CYCLE**), из текста программы убирается.
Таким образом, все команды программы, от 1-й командой ПП **CYCLE** и до команды **decfsz SecL,F**, можно считать "врезкой" в "кольцо" циклической ПП задержки **CYCLE**, которая "проходится" рабочей точкой программы за время, равное одному периоду сигнала тонального вызова.
"Старую" "врезку" (**clrwdt**) можно удалить, так как **WDT**, при формировании периода, и без этого сбрасывается часто.
Теперь можно сформировать трехсекундный интервал времени.
Период тонального сигнала равен **690 мкс.** (см. выше).
Таким образом, необходим счетчик, считающий, как минимум, до 3000000:690=4347,826 (округляем) =**4348**.
"Грубая прикидка": при применении стандартной ПП задержки (полный цикл ПП = 3 м.ц.), значение константы, записываемой в назначенный, в качестве счетчика, регистр общего назначения, должно быть 4348:3= приблизительно **1450**.
Максимальное значение константы, которую можно "заложить" в один регистр, равно **.255**
На одном регистре общего назначения, при использовании хитроумной "врезки" типа дополнительной ПП задержки, собрать такой счетчик, конечно же, можно, но это не самый лучший выход из положения (потребуется много команд).
Гораздо удобнее и проще применить **двухбайтный счетчик** (не путать с двумя однобайтными счетчиками!).
Такой счетчик считает до 256x256=**65536** и, с его помощью, можно сформировать (применительно к нашему случаю) не только трехсекундный интервал времени, но и гораздо больший.
Если речь идет о нескольких байтах, то нужно определить порядок их старшинства.
Старший байт пометим буквой **H**, а младший, буквой **L**.
Примечание: такого рода пометки - стандарт, хотя можно и придумать что-нибудь свое.
Если счетчик трехбайтный, то добавляется буква **M** - средний байт (порядок старшинства: **H, M, L**), а если четырехбайтный, то добавляются буквы **HH** - "старший старшего" (порядок старшинства: **HH, H, M, L**).
Под это дело, в "шапке" программы, "прописываем" регистры общего назначения с названиями **SecH** (в нем "лежит" старший байт) и **SecL** (в нем "лежит" младший байт) и назначаем им адреса в области оперативной памяти.
Например: **0Dh** и **0Eh** соответственно.
Вот Вам и ответ на вопрос: откуда, в "шапке" программы, взялись регистры **SecH** и **SecL**?
В части, касающейся рассматриваемой группы команд, получилось то, что Вы видите в тексте программы **cus**.
Давайте разберемся с 2-байтным счетчиком (что это такое и откуда что взялось?).
Стандартный принцип организации работы 2-байтного вычитающего счетчика:
- сначала декрементируется содержимое регистра младшего байта,

- а после его очищения (в счетчике - ноль), декрементируется содержимое регистра старшего байта.

Примечание: N-байтный счетчик может называться N-разрядным счетчиком.
В последнем случае, имеется ввиду количество байтов. Строго говоря, такое определение удобно, но не вполне корректно. Имейте это ввиду.
Далее, я буду использовать оба этих понятия. Именно так и "закаляется сталь/приобретается иммунитет" ("организмы должны быть стойкими").
Декремент счетчика младшего разряда будет происходить каждый раз после формирования периода тонального сигнала, а декремент счетчика старшего разряда будет происходить через каждые 256 периодов тонального сигнала, в момент смены, в младшем разряде счетчика, числа .255 на число .0
Сигналом окончания формирования трехсекундного интервала времени является очищение счетчика старшего разряда, после чего рабочая точка программы должна "уйти" в сценарий "программа исполняется далее".
По своей сути, этот "механизм" ничем не отличается от "механизма" работы счетчика, собранного, например, на микросхемах счетчиков (...ИЕ...) и тот, кто с ними работал, без особого труда поймет, о чем идет речь.
В регистры **SecH** и **SecL**, нужно записать какие-то константы.
Нужно определиться с их числовыми значениями.
Для того чтобы "отмерить" 3 сек., 2-хразрядный счетчик должен посчитать **4348** периодов (см. выше).
Счетчик младшего разряда декрементируется каждый период.
Если установить в нем константу, например, **.255**, то в процессе последовательного декрементирования, после установки числа **.0**, произойдет один декремент содержимого счетчика старшего разряда.
Затем, в счетчике младшего разряда, происходит переход от **.0** к **.255** и все повторяется снова. И так происходит много раз (счет "по кольцу"). Пока не обнулится счетчик старшего разряда.
Таким образом, при прохождении **4348** периодов, должно произойти примерно **17** декрементов содержимого счетчика старшего разряда.
Следовательно, для того чтобы очистить счетчик старшего разряда примерно за 3 сек., необходимо записать, в регистр **SecH**, константу **.17**
В данном случае, в счетчик младшего разряда записывается максимально возможное значение константы (**.255**), которое, по этой причине, условно можно приравнять единице значения константы старшего разряда.
Таким образом, в счетчик старшего разряда необходимо заложить "прикидочную" ("грубую") константу **.17 - .1 = .16**
Обращаю Ваше внимание на следующее.
Между первой командой ПП **CYCLE** и командой **decfsz SecL,F** располагаются не только отдельные команды, но и целых две циклические подпрограммы.
Подпрограмма **CYCLE** классифицируется как циклическая подпрограмма задержки, с "массивной врезкой", включающей в себя две циклические ПП задержки **PAUSE_1** и **PAUSE_2**.
Вот Вам и наглядная иллюстрация того, о чем я говорил ранее: в состав "врезки" могут входить не только отдельные команды, но и целые циклические подпрограммы.
В циклическую ПП задержки, образно выражаясь, можно "врезать все что угодно". Главное при этом то, чтобы после этого, ПП задержки не "потеряла свою жизнеспособность" и обеспечивала нужное время задержки.
Если говорить конкретно о циклической ПП **CYCLE**, то можно сделать следующий вывод: "врезка" подпрограммы **CYCLE** работает по заданному разработчиком алгоритму и обеспечивает полный цикл этой подпрограммы, равный периоду сигнала с частотой 1450 Гц. (**690 мкс**.).
2-хразрядный счетчик отсчитывает число периодов, "помещающихся" в трехсекундном интервале времени, и по его окончании, осуществляется переход в сценарий "программа исполняется далее".
"Для полного счастья", остается только детально разобраться с константами трехсекундного счетчика (таймера).
Чтобы излишне не мудрствовать, "привяжусь" к тем значениям констант, которые были указаны выше.

То есть, **.255** в младшем разряде счетчика и **.16** в старшем.
С учетом того, что "прикидка" производилась "грубо", и в результате этого, реальное время "выхода" в эфир сигнала тонального вызова несколько больше, чем расчетное, я на единицу уменьшил числовое значение константы счетчика старшего разряда и сделал ее равной **.15**.
То есть, в этом случае, сигнал тонального вызова будет "выдаваться" в эфир в течение времени немного меньшего, чем 3 сек.
"На фоне слова приблизительно", это приемлемо.
Лично я, решил так, а другой человек может принять другое решение.
Например, можно уменьшить числовое значение константы счетчика младшего разряда, а числовое значение константы счетчика старшего разряда не менять, или перейти к другому соотношению числовых значений констант, или вообще задать другое значение интервала времени "выхода" в эфир сигнала тонального вызова.
Если, как в данном случае, точной калибровки не требуется, то достаточно и приблизительного расчета, подобного приведенному выше.
То есть, рассчитанные ранее константы, можно "смело закладывать" в программу, и в дальнейшем, не тратить время на проверку истинного значения трехсекундного интервала времени в симуляторе.
Естественно, что в этом случае, нужно быть уверенным, что расчеты произведены правильно.
Если возникают сомнения по поводу правильности выбора числовых значений времязадающих констант, то милости просим в симулятор. Он для этого и существует.
Если же речь идет о формировании калиброванных (точных) интервалов времени (в нашем случае, времени полупериодов), то в большинстве случаев, без проверки их значений в симуляторе и соответствующей "рихтовки", не обойтись.
Итак, "окончательно и бесповоротно" назначаю: константа старшего разряда счетчика = **.15**, константа младшего разряда счетчика = **.255** .
Теперь возникает вопрос: "В какое место текста программы нужно врезать группу команд записи констант"?
Ответ на этот вопрос был дан ранее, и поэтому "врезаю" эту группу команд в концовку ПП **START**.
Вот Вам и ответ на вопрос: "Откуда взялась, в концовке ПП **START**, группа команд записи констант и почему она выглядит именно так"?
Если производится последовательная запись констант в несколько регистров, то порядок этой записи не имеет значения.
Применительно к данному случаю, это означает то, что сначала можно записать константу в регистр **SecH**, а потом, в регистр **SecL** или наоборот.
Что получается? А получается это:
Декремент содержимого 2-хразрядного счетчика производится при наличии, на НЧ выходе устройства, единичного уровня, так как рабочая точка программы "заходит в этот счетчик" во время формирования положительного полупериода.
Во время формирования 3-хсекундного интервала времени, после перехода рабочей точки программы в ПП **CYCLE**, установка нулевого уровня, на НЧ выходе устройства, происходит после исполнения команды **bcf PortB,2**.
Далее, с помощью ПП **PAUSE_1**, этот уровень фиксируется на время отработки полупериода.
Далее, после исполнения команды **bsf PortB,2**, на НЧ выходе устройства, устанавливается единичный уровень.
Далее, с помощью ПП **PAUSE_2**, этот уровень фиксируется на время отработки полупериода.
Далее, происходит декремент содержимого 2-разрядного счетчика.
Если результат декремента не равен нулю, то происходит переход в ПП **CYCLE** (**goto CYCLE**) и все повторяется снова и снова, до тех пор, пока, с момента включения на передачу, не пройдет примерно 3 сек.
Посмотрите в текст программы **cus**.
После 4-х команд 2-хразрядного счетчика, Вы видите команду **bcf PortB,2**
Именно на эту команду переходит рабочая точка программы после того, как счетчик старшего разряда очистится, то есть, завершится формирование 3-хсекундного интервала времени "выхода" тонального сигнала в эфир.
Этой командой осуществляется смена единичного уровня, на НЧ выходе устройства, на нулевой.

Это сделано для того, чтобы после очищения счетчика старшего разряда, закончить формирование положительного полупериода.
Собственно говоря, эта команда вставлена в текст программы "для порядка" и без нее вполне можно обойтись (убрать).
И в самом деле, какая разница, какой уровень "выставлен" на НЧ выходе устройства после формирования 3-хсекундного интервала времени "выхода" тонального сигнала в эфир (см. разделительный конденсатор)?
Эту команду можно оставить, а можно и удалить.
Далее, рабочая точка программы "влетает" в ПП PRD, которую сейчас предстоит создать.
Напоминаю, что она должна делать.
В этой ПП, рабочая точка программы должна уйти в "вечное кольцо" и выйти из него в момент переключения с передачи на прием, с последующим переходом на новый цикл программы.
"Механизм" функционирования "вечного кольца" рассмотрен ранее (см. "вечное кольцо" ПП **START**).
И в ПП **PRD** он точно такой же.
Разница только в деталях: так как "закольцовка" происходит внутри ПП **PRD**, то команда безусловного перехода должна обращаться к ПП **PRD**, и при опросе состояния вывода **RB0,** вместо команды **btfsc**, используется команда **btfss** (если применить **btfsc**, то будет "вилка").
Если речь идет о длительной "закольцовке" (что и имеет место быть), то обязательно нужно периодически сбрасывать **WDT**.
В "вечном кольце" ПП **START**, это делалось с помощью команды **clrwdt**, расположенной в группе команд подготовительных операций.
Примерно так же нужно поступить и по отношению к ПП **PRD**.
То есть, нужно "врезать" команду **clrwdt** в цикл "вечного кольца" этой подпрограммы.
В данном случае, команду **clrwdt** можно "пристроить" единственно разумным способом - сверху от команды ветвления, с переносом названия подпрограммы **PRD** на команду **clrwdt**.
Посмотрите, как это выглядит в тексте программы **cus**.
Необходимость применения команды **clrwdt** обусловлена тем, что время включения на передачу может быть бОльшим, чем 2,3 сек.
Первые 3 команды ПП **PRD** можно классифицировать так: циклическая ПП задержки, с "врезкой" из одной команды, с "уходом в вечное кольцо" и с выходом из него по внешнему управляющему сигналу.
Для сценария типа "закольцовка", цикл такой подпрограммы равен 3 м.ц.+1 м.ц. команды **clrwdt** = 4 м.ц. и поэтому срабатывания **WDT** опасаться не нужно.
Итак, мы дошли до конца полного цикла программы.
Остается только вставить в текст программы команду безусловного перехода на "новый", полный цикл программы (**goto START**), чем и обеспечивается соблюдение "глобального" принципа: ***рабочая точка программы всегда должна находиться в движении.***
Текст программы всегда заканчивается директивой **end** для того, чтобы **MPLAB** "знал", что ниже этой директивы, текст программы "искать" не нужно.
Замечание по **WDT**: естественно, что длительность импульса, вырабатываемого **WDT** (в нашем случае, **2,3 сек**.), можно уменьшить за счет уменьшения коэффициента деления предделителя.
Можно отключить предделитель от выхода **WDT** (18-ти мс. вполне достаточно) и даже вообще отключить **WDT** (в битах конфигурации).
Если имеется такое учебно-тренировочное желание, то нужно изменить числовое значение константы, записываемой в регистр **OptionR** или изменить числовое значение слова конфигурации.
Предполагаю, что после прочтения изложенной выше информации, возможно возникновение "неразберихи" в терминологии, связанной с такими понятиями как: подпрограмма задержки, счетчик, "уход в вечное" кольцо и т.д.
Все это, в своей основе, одно и то же, хотя, казалось бы, речь идет о выполнении разных функций.
И в самом деле, что общего между, например, ПП **PAUSE_1**, ПП **PRD** и ПП **CYCLE**, кроме того, что все они - циклические?
Конструкции разные, и выполняемые функции также разные.
Однако, в основе "всего этого великолепия", лежат такие понятия как "задержка" и "счетчик" ("сиамские близнецы").
Например, подпрограмму **PRD** можно классифицировать как подпрограмму задержки, с

выходом из нее по внешнему воздействию, но ее можно считать и счетчиком, который отсчитывает время, от момента очистки старшего разряда трехсекундного счетчика и до момента перехода с передачи на прием. И первое, и второе "имеют право на жизнь".
Примерно то же самое (с поправкой на специфику) можно сказать и о любых других циклических подпрограммах.
Такого рода классификация - процесс субъективный, и поэтому одну и ту же подпрограмму можно классифицировать по-разному.
Это зависит от того, какое из двух этих восприятий наиболее удобно (кто-то видит счетчик, а кто-то видит задержку или их "гибрид").
Частенько из-за этого возникают "разнобои", которые не способствуют взаимопониманию.
Конечно, если человек "упертый", то он, через "пинки, синяки и шишки", рано или поздно выработает в себе умение смотреть на любую циклическую подпрограмму с обеих точек зрения (счетчик/задержка), но с целью минимизации указанных выше "бяк", выгоднее изначально разобраться с подобного рода "вещами" и не жалеть на это времени.
В качестве тренировки, попробуйте разглядеть в любой из циклических подпрограмм программы **cus**, одновременно, и устройство типа счетчика, и устройство, осуществляющее задержку.

Дополнение

Какую именно группу команд считать подпрограммой - определяет программист.
Программист может видоизменить исходную подпрограмму так, что и "мама родная не узнает", вплоть до того, что вообще лишить ее этого "статуса", и в "статусе" группы команд, включить ее в состав другой подпрограммы.
Короче, "полная демократия".
Однако, существуют определенные правила, которых, с точки зрения "наведения элементарного порядка", нужно придерживаться.
Если имеется группа команд с четко выраженной функциональностью, то прежде чем присваивать ей "статус" подпрограммы, нужно ответить на вопрос: "Имеется ли в тексте программы хотя бы один переход на начало исполнения этой процедуры или таких переходов нет"?.
Если хотя бы один такой переход есть, то этой группе команд присваивается "статус" подпрограммы, а если таких переходов нет, то этого делать не нужно (хотя и можно), так как "захламлять" текст программы неработающими названиями подпрограмм, нет смысла.
Например, 2-х разрядный счетчик, реализованный в программе **cus**, является функционально законченным устройством и ему, формально, вполне можно "присвоить статус" подпрограммы.
В этом случае, в 1-м столбце текста программы, "прописывается" название подпрограммы (например, **XYZ**) → **XYZ decfsz SecL,F**.
Но это название не будет работать, так как в программе **cus**, отсутствует необходимость в безусловном переходе на **XYZ**.
Получается что-то типа "лишнего балласта".
А раз это так, то и незачем назначать такое неработающее название.
Что касается меток, то они применяются для того чтобы перейти на любую другую команду текста программы, которая не является первой командой подпрограммы.
Пример "манипуляций" с меткой (правда, их результат функционально "забракован", но на "технологию" работы с метками это не влияет), был приведен выше.

Краткий, общий итог

Выше описан процесс конструирования конкретного устройства на основе ПИКа.
Надеюсь на то, что после знакомства с этой информацией, Вы поняли, что если заранее определены исходные данные, то процесс составления программы не есть что-то суперсложное.
Изначально, с непривычки, некоторые из Вас могут с этим не согласиться, но после того, как Вы "набьете руку", то сами удивитесь, как это, в сущности, относительно просто.
Добрый совет: на данной стадии обучения, не стОит "стрелять из рогатки по медведю".
То есть, вовсе не нужно пытаться сразу же сконструировать что-то свое.
Для этого, как минимум, нужна "рогатина", а еще лучше, "ружье".

На первых порах, для приобретения "солидного инструментария" и его "шлифовки", советую Вам обратить самое пристальное внимание на "разбор полетов", а именно, на анализ текстов "чужих" программ.
То есть, программ для устройств с готовыми исходными данными (неопределенности отсутствуют).
В этом, я, по мере своих возможностей, постараюсь эффективно Вам помочь.
В том смысле, что кроме подробного анализа текстов программ, я постараюсь показать, как именно происходит "борьба с неопределенностями", вразумительное описание чего, по совокупности причин, является большой редкостью.

В конечном итоге, когда Вы "набьете руку" и займетесь практическим конструированием, "центр сложности" несколько сместится, из "технологического сектора работы", в "сектор работы по борьбе с неопределенностями".
До этого еще далековато.
Сейчас же, главная задача → "прочувствовать (в том числе, и на уровне подсознания) правила игры". Пока, в приложении к простым программам.
Ранее, в ходе работы над текстом программы, "механизм" визуального, Образного восприятия работы программы (не все в нашем деле определяется только прямолинейной логикой) был задействован как-то слабовато.
Теперь пора его "включить на полную мощность".
Речь идет о работе в симуляторе.
Кроме того, что Вы реально увидите, как рабочая точка программы движется по тексту программы (как исполняется программа и что при этом происходит в "недрах" ПИКа), Вы узнаете и то, что такое отладка программы.

9. Работа в симуляторе. Отладка программы.

О том, что такое симулятор, говорилось ранее. Подошло время "привести этот мощнейший инструмент в полную боевую готовность".

Введение

Для создания общего представления о том, зачем нужен симулятор, достаточно представить себе разницу, например, между счётами и компьютером.
Естественно, что бухгалтер, работающий на компьютере, сделает одну и ту же работу быстрее и качественнее, чем если бы он работал со счетами.
Не может также вызвать сомнения и то, что прежде чем быстро и качественно сделать работу, необходимо, сначала, хотя бы знать, "с какого бока подойти" к этому самому компьютеру, на что нажимать и что после этого будет происходить.
Первичное представление о работе в симуляторе можно составить, усвоив соответствующую информацию из раздела, посвященного **MPLAB**.
В дальнейшем, я исхожу из предположения, что эта информация Вами усвоена.
Руководствуясь принципом "от простого к сложному", сначала рассмотрим случай:
программа, с которой работает симулятор, заведомо работоспособна. То есть, заданный разработчиком алгоритм ее работы, безошибочно выполняется.
Обращаю Ваше внимание на то, что безошибочно выполняемый алгоритм работы программы означает только то, что выполнение программы происходит в строгом соответствии с блок - схемой программы, и все ее "квадратики", в тексте программы, реализованы без ошибок функционального характера, но не то, что эта программа, в части касающейся точного соблюдения, заданных разработчиком, временнЫх характеристик устройства, идеальна.
Например, имеется работоспособный частотомер, но из-за того, что в тексте программы неверно подобраны числовые значения времязадающих констант или использован кварц с номиналом частоты, отличающимся от рекомендованного, он будет "врать".
Если он "врет не сильно", то можно "превратить" его в "полноценный" измерительный прибор, корректируя значение частоты кварцевого генератора.
А если он "врет сильно" и эта коррекция не эффективна?
А если, при переходе с одного интервала времени измерения на другой, существует большая разница в показаниях одного и того же значения частоты?
В этих случаях, нужно "лезть" в текст программы и корректировать значения времязадающих констант или "подгонять" время исполнения одного сценария работы программы под время исполнения другого сценария (я это называю "выравниванием сценариев").
Сделаем это, пока, теоретически и в общем виде, без "привязки" к какой-то конкретной программе.
Итак, "запускаем" **MPLAB** и открываем проект с программой "врущего" частотомера.
Допустим, что перед нами → текст программы, созданной под частотомер.
Естественно, что сначала, в тексте программы, нужно найти группы команд "закладки" соответствующих времязадающих констант, в регистры общего назначения, созданные и "прописанные" в "шапке" программы под это дело.
Эти группы команд - стандартные (о них говорилось выше: пары команд: **movlw** и **movwf**) и поэтому обнаружить их не составляет большого труда.
Если имеются несколько отдельных констант и/или групп констант (такое бывает часто), то необходимо выяснить, в реализации каких именно функциональностей они задействованы, и анализируя текст программы, "отсеять" ненужные.
Кстати, то, что Вы сейчас читаете, это примитивный пример использования приемов программно-аппаратного анализа.
В части касающейся "тренировки мозгов", такая работа не то что полезна, а исключительно полезна.
Естественно, что такого рода "манипуляции" предполагают наличие опыта.
Если он есть, и человек владеет приемами программно-аппаратного анализа, то даже при отсутствии комментариев к тексту "чужой" программы, в ней, по большому счету, можно "вычислить" все что угодно, и что-то целенаправленно изменить.
В конце концов, "материнскую" программу можно "перекроить так, что мама родная не узнает" или, на ее основе, создать нечто новое.
Именно к этому нужно всячески стремиться, и именно это является одним из главных признаков классного программиста, который, по определению (по умолчанию), является

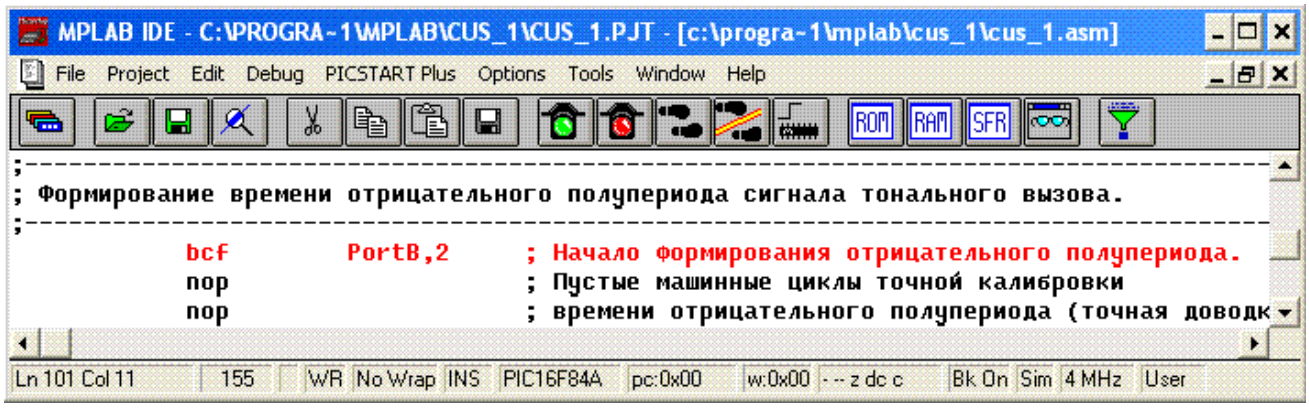
хакером, "взломщиком" (или еще какой-нибудь "редиской").
Примечание: этим "делам" будет посвящена целая книга.
Предположим, что место "закладки" констант обнаружено, и их числовые значения известны.
Если речь идет только об одной константе, то работаем только с ней.
А как быть, если их, например, 3 или 4 (кстати, программы под реальные частотомеры и другие более-менее сложные устройства, содержат примерно такое же количество констант, определяющих величину измерительного интервала времени)?
В этом случае, нужно определить порядок старшинства.
В большинстве текстов программ, порядок старшинства стандартный, и он "привязан" к конкретным буквам.
Просто посмотрите на последние буквы названий регистров общего назначения, в которые записываются константы (вспоминайте: **HH, H, M, L** или нечто подобное).
Например, регистры **SecH** и **SecL** программы **cus**.
Одиночная константа, в большинстве случаев, записывается в регистр общего назначения, название которого этих букв не содержит (например, регистр **Sec** программы **cus**).
Порядок старшинства можно также определить, анализируя положения регистров в группе команд счетчика (в сАмом верху – регистр самого младшего разряда , а в сАмом низу – регистр самого старшего разряда).
Зная порядок старшинства и оценивая величину погрешности в показаниях частотомера, можно определиться, в каком (каких) из этих разрядов следует произвести числовую коррекцию константы (констант).
Если погрешность в показаниях частотомера маленькая, то можно откорректировать только числовое значение константы младшего разряда, а если она значительна, то вдобавок к этому, придется корректировать и числовые значения констант более старших разрядов.
"Технология" такой коррекции очень проста: в текстовом редакторе **MPLAB**, одно число заменяется на другое, после чего производится ассемблирование.
Возникает вопрос: "В какую сторону производить изменения: в сторону увеличения или в сторону уменьшения"?
Предположим, что частотомер показывает значение частоты меньшей, чем истинное и "врет" не сильно.
Следовательно, нужно изменить числовое значение константы младшего разряда.
В какую сторону?
Прежде чем ответить на этот вопрос, нужно кое с чем разобраться.
Предположим, что обнаружены "места закладки" констант трех разрядов (H, M, L).
Запоминаем названия регистров общего назначения, в которые "закладываются" эти константы, и ищем, в тексте программы, команды, которые обращаются к содержимому этих регистров.
Это неизбежно приведет к обнаружению (идентификации) группы команд трехразрядного счетчика, в состав которой будут входить команды **incfsz** или/и **decfsz**.
В данном случае (нужно увеличить интервал времени измерения), нужно посмотреть, какая именно, из этих двух команд, обращается к содержимому регистра общего назначения самого младшего разряда (L).
Если это команда **decfsz** (в большинстве случаев), то величину константы младшего разряда счетчика нужно увеличивать, а если **incfsz**, то уменьшать.
Если для того чтобы устранить погрешность показаний частотомера, диапазона изменений значения константы счетчика младшего разряда не достаточно, то нужно изменить значение константы счетчика среднего разряда (M).
Если и это не приводит к успеху, то нужно изменить значение константы счетчика старшего разряда (H).
Если корректируется числовое значение константы, например, счетчика среднего разряда (коррекция типа "грубо"), то в большинстве случаев, после нее, необходимо произвести коррекцию числового значения константы счетчика младшего разряда (коррекция типа "точно").
В большинстве случаев (но не во всех), "манипуляций" с числовыми значениями констант счетчиков среднего и младшего разрядов, оказывается достаточным.
Если требуется сформировать калиброванный интервал времени с максимально возможной точностью (с точностью до 1-го машинного цикла), а "шаг" изменения константы счетчика младшего разряда этого сделать не позволяет (вспомните, что минимальный цикл ПП задержки составляет 3 м.ц., а для подпрограмм с "врезками", он еще больше), то нужно либо

"врезать", в текст программы, один или несколько **NOP**ов, либо "вырезать", из текста программы, один или несколько **NOP**ов (если они имеются в наличии).
Эти **NOP**ы "врезаются", в текст программы, либо до группы команд счетчика, либо после нее (то есть, в непосредственной близости от этой группы команд).
Посмотрите в текст программы **cus**.
В группах команд, формирующих время отрицательного и положительного полупериодов, Вы увидите группы **NOP**ов.
Это как раз то, о чем шла речь.
Точную калибровку можно обеспечить и путем изменения времени отработки одного цикла ПП задержки.
Например, можно увеличить его с 3-х м.ц., до 4-х или 5-ти (и т.д) м.ц.
Иногда это помогает, но в большинстве случаев, нужен комплексный подход.
То, что Вы прочитали, есть теория одного из вариантов отладки программы (вернее, "взлома", с последующей отладкой), для случая, когда необходимо только изменить временнЫе характеристики устройства или "подогнать" их, в процессе конструирования, под задуманное.
Как это делается на практике?
А вот так:

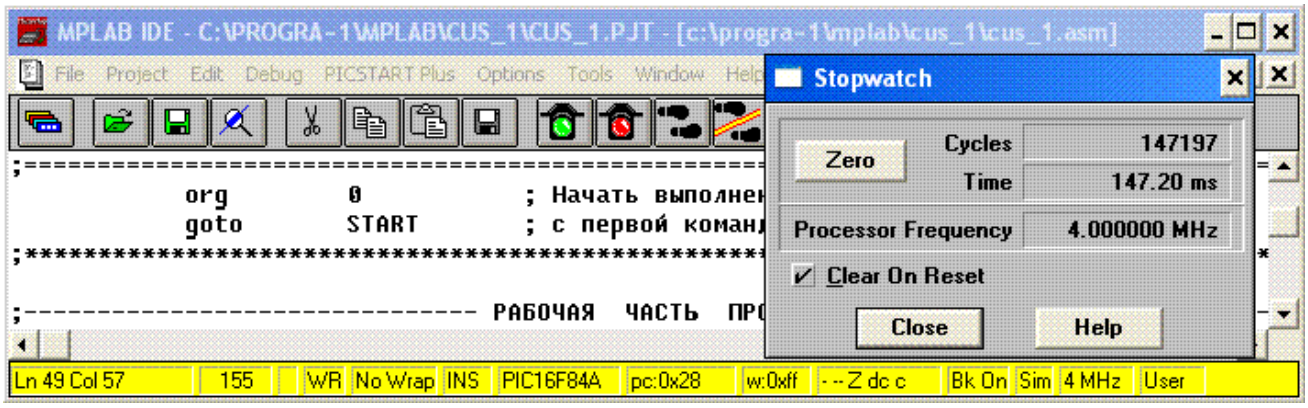
Отладка временных характеристик устройства, обслуживаемого программой cus

В отличие от теоретического случая, описанного выше, в программе **cus**, "взламывать" что-то совсем не нужно, так как в "конструкции" этой программы, никаких "секретов" нет (я надеюсь на это).
То есть, "вычислять места закладок" констант и определять "конструкцию" счетчиков, в которых они используются, не нужно.
Если предположить, что программа **cus** работоспособна (а так оно и есть) и величину трехсекундного интервала времени корректировать не нужно (его корректировать не нужно), то ее отладка сводится к точной коррекции значений интервалов времени отрицательного и положительного полупериодов.
Они должны быть одинаковыми (форма сигнала - "меандр") и равными 345 мкс. или 345 машинным циклам (кварц 4 мГц.).
Откройте проект **cus** (любой: **cus_1** или **cus_2**).
Ранее говорилось, что в состав симулятора входит **секундомер**.
Это как раз тот случай, когда он архинужен.
Чтобы что-то им измерить, нужно знать что измерять (количество машинных циклов формирования полупериодов) и определить "границы" замеров.
Если не сделать последнего, то можно просидеть перед компьютером хоть целый день, так и не дождавшись конца подсчета.
Таким образом, речь идет о назначении **точек остановок**.
Технически, назначить их очень просто (об этом говорилось ранее).
Сложность заключается в том, **какие именно команды назначать точками остановок?**
Для того чтобы не было проблем с назначением точек остановок, программист должен либо "свободно ориентироваться" в тексте программы, либо, как минимум, знать все "ключевые точки" этой программы.
Для программы **cus**, такими "ключевыми точками" являются команды **bcf PortB,2** (начало формирования отрицательного полупериода) и **bsf PortB,2** (начало формирования положительного полупериода).
Примечание: еще одна команда **bcf PortB,2** , расположенная ниже, по причинам, указанным в предыдущем разделе, к этим "ключевым точкам" не относится.
И в самом деле, если необходимо измерить время формирования полупериода, его нужно мерить от перепада до перепада.
Итак, **точки остановки** четко определены. Это команды **bcf PortB,2** и **bsf PortB,2**.
Перед началом практической работы, необходимо уяснить следующую,"техническую деталь".
Если в ходе работы над текстом программы (любой), появляется необходимость в его редактировании, а после этого редактирования, нужно сохранить в проекте исходный (начальный) текст программы, то при закрытии проекта, когда Вам будет предложено ответить на вопрос: ***Сохранить изменения или нет?***, необходимо нажать на ***No***.

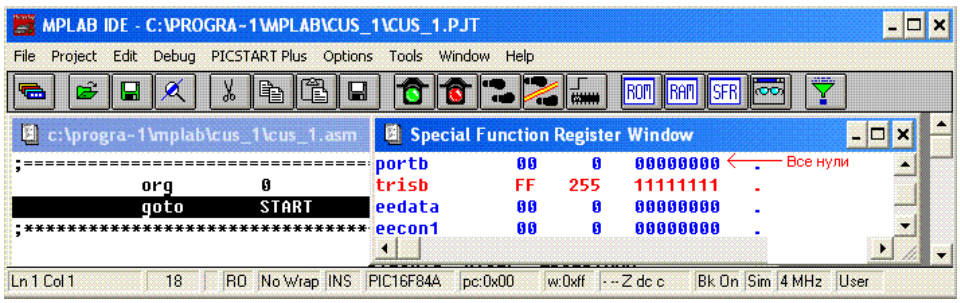
Напоминаю Вам, что в обучающих целях, предполагается, что программа **cus** составляется "с чистого листа".
То есть, в регистр **Sec**, в обоих случаях (как при формировании отрицательного, так и при формировании положительного полупериодов), записываются ранее рассчитанные константы ("грубо"), и **NOP**ы "точной" калибровки отсутствуют.
Сделаем так: сначала, для уяснения "технологии" замеров, будем работать с полностью отработанным текстом программы, а затем, для иллюстрации процесса отладки, соответствующим образом изменим текст программы (уберем **NOP**ы и заменим значения констант на расчетные) и произведем, после этого, ее отладку.
Обратите внимание на то, что "ключевые команды" не являются первыми командами ни одной из подпрограмм, и они не помечены (меток нет).
Следовательно, назначить их точками остановок, в окне ***Break Settings*** (вспоминайте: ***Debug → Break Settings...***) не получится (можете заглянуть в это окно и убедитесь в этом).
Таким образом, точки остановок будем назначать, щелкая по строкам с этими командами (кто забыл → вернитесь назад и прочитайте, как это делается).
Перед замером, все команды программы, находящиеся в тексте программы до первой точки остановки (верхней), желательно исполнить.
Итак, первую (верхнюю) точку остановки нужно выставить на команде **bcf PortB,2**
Выставляем ее, щелкнув правой кнопкой мыши по строке с этой командой и выбрав, в выпадающем списке, ***Break*** :



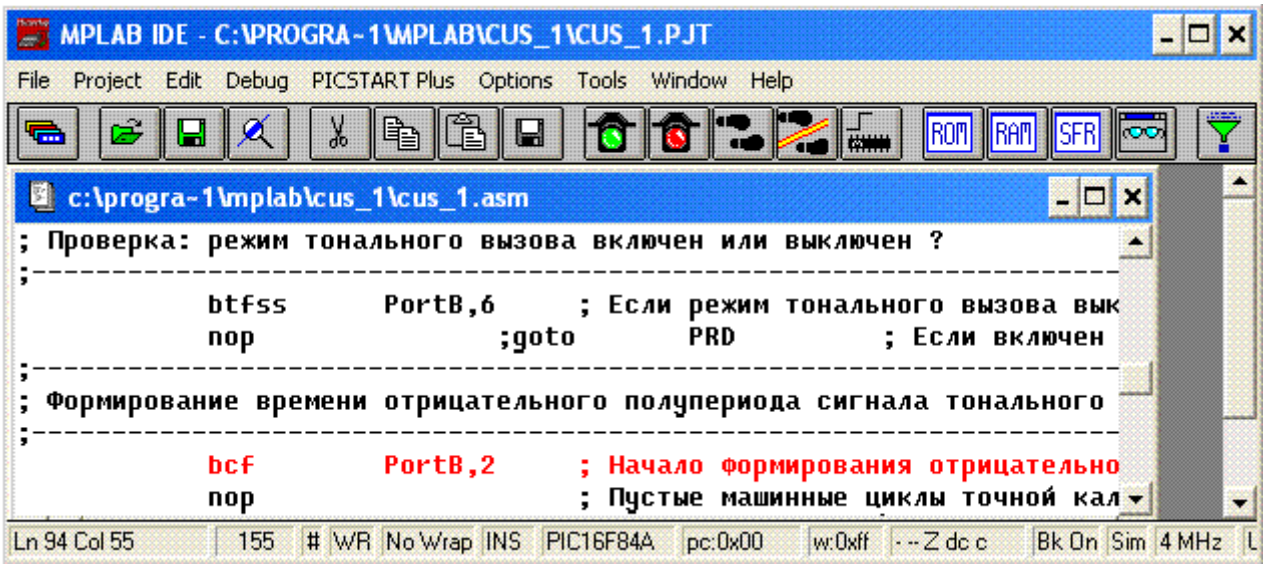
Содержимое строки окрасилась в **красный** цвет, и теперь видно, что эта команда является точкой остановки.
Теперь нужно "добраться" до этой команды, что предполагает исполнение всех предшествующих команд программы, начиная с первой команды ПП **START**.
Можно сделать это и вручную, щелкнув по кнопке сброса программы на начало, а затем, щелкая по кнопке со следами, а можно "добраться" до этой команды и в "автомате", что гораздо быстрее.
Сделаем последнее:
Щелкаем по кнопке сброса программы на начало, после чего Вы увидите, что рабочая точка программы перешла на ее начало (**goto START**).
Вызываем окно секундомера (***Window - Stopwatch***).
Убедитесь, что в квадратике ***Clear On Reset*** , галочка установлена.
Секундомер должен показывать нули.
Щелкнув по кнопке с **зеленым** светофором, запускаем "автомат" (автоматическое исполнение программы).
В идеале, через некоторое время, рабочая точка программы должна остановиться на назначенной нами точке остановки.
Во время отработки "автомата", внизу окна **MPLAB,** должна высветиться **желтая** полоса, которая исчезнет в момент перехода рабочей точки программы на команду, назначенную точкой остановки (вернее, точкой конца замера).
В данном случае, этого не происходит: **желтая** полоса будет высвечиваться постоянно и также постоянно показания секундомера будут увеличиваться:



Можете даже и не надеяться на остановку, так как **программа "зависла" в симуляторе**.
То есть, рабочая точка программы "гоняет в ней по кольцу" и не может из него выйти, а соответственно, и "дойти" до точки остановки.
Так что, не все в этом деле так просто. Давайте разбираться.
Вопрос: "Каким образом преодолеть это затруднение и добраться до точки остановки"?
Для этого, прежде всего, необходимо ответить на вопрос: "Если программа обеспечивает нормальную работу реального устройства, то по каким ехидным причинам она зависла в симуляторе?"
Ответ: причина простая → **не сымитированы внешние воздействия**.
То есть, либо на входе управления **RB0**, либо на входе управления **RB6**, либо в комплексе, установлен такой уровень (уровни), который "заводит" рабочую точку программы в "вечное кольцо" (либо в ПП **START**, либо в ПП **PRD**).
Поочередно пощелкайте по кнопкам с **зеленым** и **красным** светофором, и в большинстве случаев, Вы обнаружите остановившуюся рабочую точку программы (после щелчков по **красному** светофору) в подпрограмме **PRD**.
Вот Вам и то "место" программы, в котором рабочая точка "ушла в вечное кольцо".
Логически рассуждая далее, можно "вычислить" причину этого "конфуза".
Кстати, это архиполезно. Кто желает, тот может это сделать прямо сейчас, "отложив дальнейшее чтение в сторону".
Причину, по которой рабочая точка программы "не может добраться" до точки остановки, можно обнаружить очень просто, даже не прибегая к приведенным выше рассуждениям.
Сбросьте программу на начало и перейдите к пошаговому ее исполнению.
То есть, пощелкайте по кнопке со следами (или по **F7** на клавиатуре).
Одновременно следите за движением рабочей точки (то есть, за исполнением программы) в текстовом редакторе **MPLAB**.
Через несколько щелчков, Вы заметите, что после исполнения команды **goto PRD**, рабочая точка программы "улетит" в ПП **PRD**, в которой и "уйдет в вечное кольцо".
Такое может быть только в единственном случае: когда на выводе **RB6** установлен **0**.
Имейте ввиду: ***мы работаем не с реальным (физическим) устройством, а с "виртуальным". То есть, с тем, что создано (просимулировано) "внутри" MPLAB.***
Следовательно, "корень зла" нужно искать в симуляторе.
Давайте посмотрим, что, по умолчанию, выставлено симулятором на выводах порта B.
Сбросьте программу на начало. Щелкните по кнопке ***SFR***.
В окне ***Special Function Register Window***, обратите внимание на содержимое регистра **PotrB** → по умолчанию выставлены все нули (это же относится и к **PortA**).

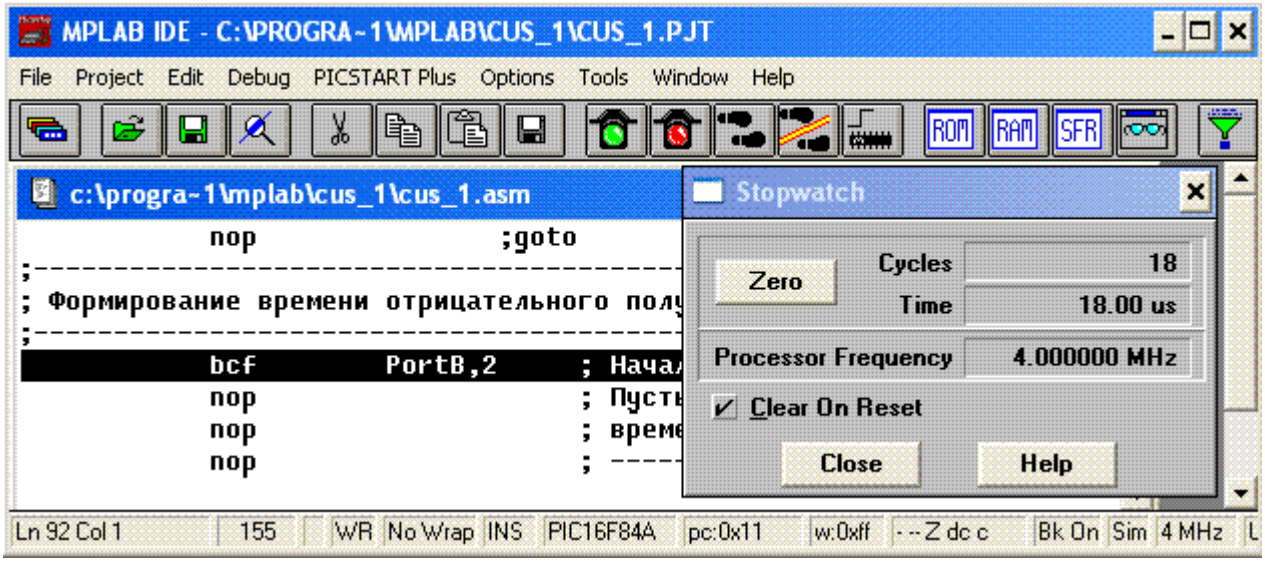


И сразу же все "становится на свои места": если **0**-й и **6**-й биты регистра **PortB** равны 0, то рабочая точка программы должна "проскочить" первую проверку (передача вкл/выкл), не "закольцевавшись" в ПП **START**, но в результате следующей проверки, она обязательно "уйдет в вечное кольцо" ПП **PRD**, где и будет "гонять по кольцу до скончания века", что мы и наблюдали.
Напрашивается естественный вывод: необходимо каким-то образом, в **6**-м бите регистра **PortB**, выставить **единицу**, и тогда все будет в порядке.
В **MPLAB** это вполне можно сделать, если задействовать так называемые **функции стимула** (асинхронный стимул, стимул порта ввода-вывода и стимул регистра. На выбор).
Это "вещь" конечно хорошая, и в будущем, я постараюсь объяснить, что это такое, но если сейчас я буду "забивать Вам этим голову", то опасаюсь, что Вы просто запутаетесь и "отклонитесь от генеральной линии", что совсем не есть хорошо.
Поступлю проще и эффективнее.
Помните, я ранее говорил о том, что при работе в симуляторе, существуют некоторые **"уловки"**?
Они "обманывают" **MPLAB** в том смысле, что позволяют обойтись без задействования функций стимула ("подменяют" их).
В данном случае (всяческих "уловок" много), команда, с которой начинается "бардак", из текста программы исключается (блокируется).
На практике, это делается так.
Проблемы создает команда **goto PRD**.
Именно она направляет рабочую точку программы в "вечное кольцо".
Заменяем эту команду на команду **nop**.
Так как позднее, команду **goto PRD** нужно будет опять "вернуть на свое место" (а **nop**, соответственно, убрать), то удобнее это сделать так: непосредственно перед командой **goto PRD,** нужно поставить точку с запятой (после этого **MPLAB** перестает "видеть" эту команду) и пробелами сместить всю строку вправо (в "район" комментария), освободив "место" для команды **nop**, после чего, в том "месте", где ранее "дислоцировалась" команда **goto PRD**, нужно "настучать" команду **nop**:



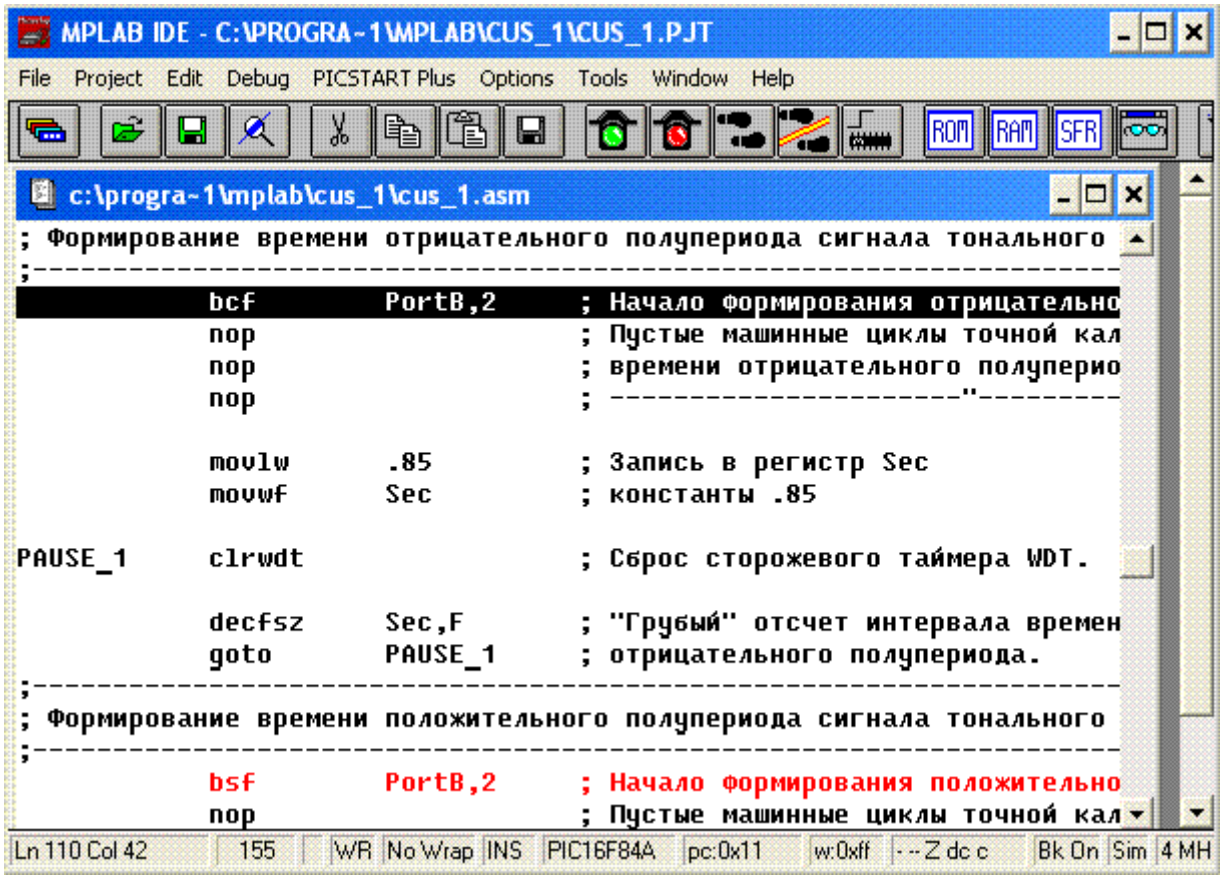
Позднее, **MPLAB** все-равно "заставит" Вас проассемблировать текст программы, так что лучше сразу же после внесения изменения (любого) в текст программы (а мы его внесли), произвести ассемблирование.
Заодно, можно убедиться в отсутствии ошибок (Вы получите сообщение о безошибочном ассемблировании).
Внимание: ***после ассемблирования, точки остановки снимаются и их нужно переназначить*** (назначить по-новой).
Что получилось?
А получилось то, что после выполнения команды **btfss PortB,6**, рабочая точка программы, через **nop** , "встанет" на команду **bcf PortB,2**, и ее "ухода, в вечное кольцо" ПП **PRD**, не

произойдет.
А раз это так, то можно без проблем "добраться" до назначенной точки остановки.
Далее, возникает вопрос: "Если команда **goto** исключена из текста программы, то не отразится ли это на значениях калиброванных интервалов времени полупериодов?"
Ответ: в данном случае, не отразится.
Объяснение: не смотря на то, что **6**-й бит регистра **PortB** установлен в **0**, реально исполняется (имитируется) сценарий работы, для случая установки **6**-го бита регистра **PortB** в **1**. И в самом деле, рабочая точка программы переходит на команду **bcf PortB,2** через **nop**, но только этот **nop** не "виртуальный" (см. объяснение работы команд ветвления, которое было дано ранее), а реальный.
Проще говоря, "виртуальный" **NOP** заменен на реальный ("что в лоб, что по лбу, все едино").
Теперь можно, без проблем, "добраться" до назначенной точки остановки, как в "автомате", так и пошагово исполняя программу ("добираться" совсем не долго).
Например, в "автомате": сбросьте программу на начало и щелкните по кнопке с **зеленым** светофором.
После отработки "автомата", Вы увидите, что рабочая точка программы "встала" на назначенную точку остановки (**bcf PortB,2**), что и требуется:

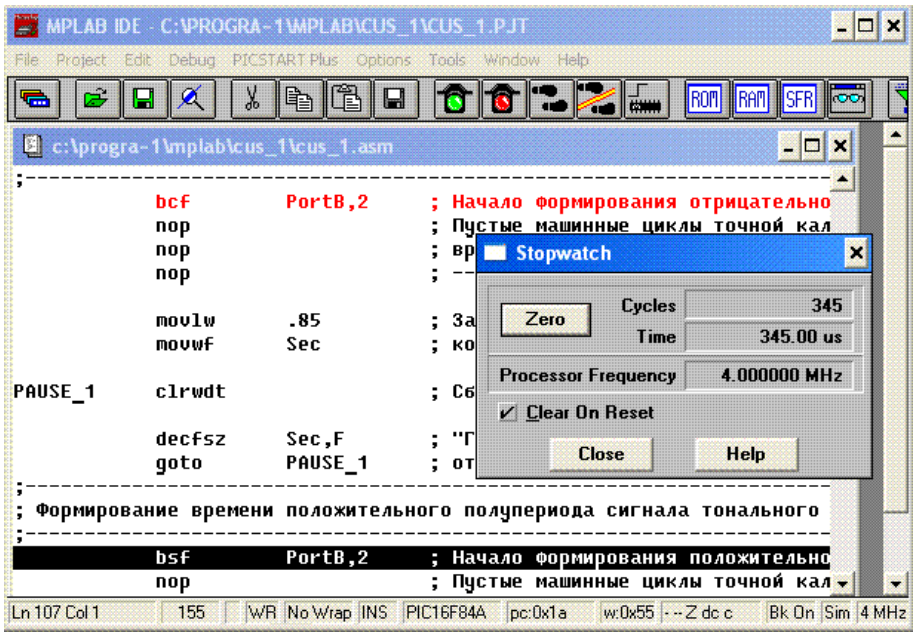


Итак, мы "вышли" на начало замера.
Посмотрите на показания секундомера.
Вы видите количество машинных циклов, отработанных от начала программы и до установки рабочей точки программы на верхнюю точку остановки (точку начала замера) и их перевод во время.
Теперь понаблюдайте, что происходит при пошаговом исполнении программы: сбросьте программу на начало и пощелкайте по кнопке со следами или по клавише **F7**.
Обратите внимание на процесс счета машинных циклов секундомером.
Вы увидите, что при выполнении команд, исполняющихся за 1 м.ц., показание счетчика машинных циклов будет увеличиваться на 1, а при выполнении команд, исполняющихся за 2 м.ц., показание счетчика будет увеличиваться на 2.
В будущем, если у Вас возникнут затруднения в определении количества машинных циклов исполнения той или иной команды или группы команд (что более актуально), то их можно устранить, пошагово исполняя команду (команды) и сверяясь с показаниями счетчика.
Обратите также внимание на то, что при пошаговом исполнении программы, установленные точки остановки игнорируются.
Это указывает на то, что ***точки остановки работают только в "автомате".***
Вывод: описанный выше, простенький прием, позволяет отказаться от использования достаточно трудоемких функций стимула и существенно упростить "выход" рабочей точки программы на точку остановки.
Таким образом, **производя несложные манипуляции (с учетом специфических особенностей той или иной команды), можно направить движение рабочей точки программы в нужный для программиста сценарий.**

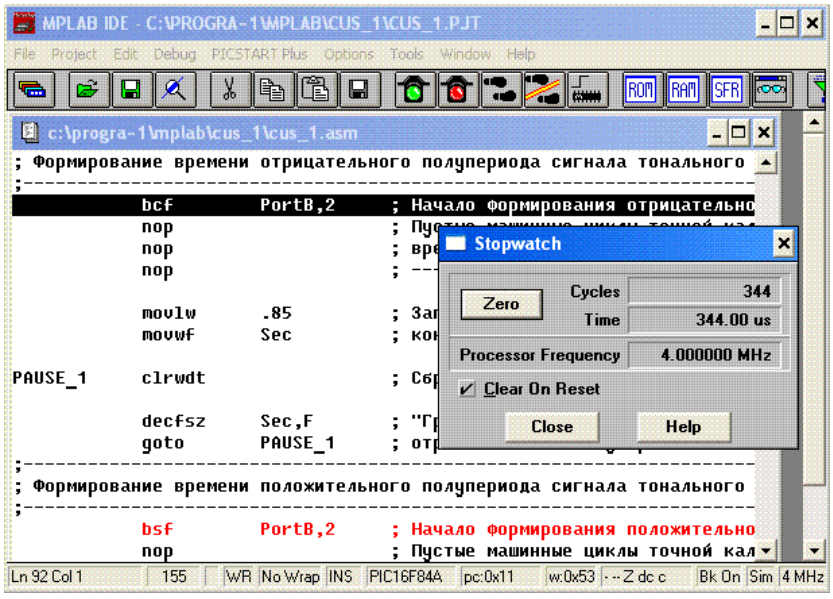
Естественно, что при этом нужно знать, как работает программа и четко представлять себе "механизм" работы той или иной команды (особенно, команд ветвления).
Лично я, прибегаю к помощи функций стимула очень и очень редко и в основном, использую "уловки". И это вовсе не есть проявление лени, а скорее всего наоборот.
Такого рода "тренировка извилины" исключительно полезна.
Еще раз напоминаю: на команду, назначенную точкой остановки, можно "выйти" 2-мя способами:
Первый: в режиме пошагового исполнения программы.
При этом, точка остановки не назначается (какой смысл? Она не работает).
Этот режим применяется при небольшом количестве команд, которые нужно отработать.
В случае, если эта группа команд "объемиста" или в ее состав входит (входят) ПП задержки со значительным временем задержки, то "выход" на точку остановки лучше делать в "автомате".
Второй: в "автомате".
В этом случае, обязательно нужно назначить хотя бы одну точку остановки, но можно и больше.
Итак, мы "стоИм" на верхней точке остановки (**bcf PortB,2**), то есть, на той команде, после исполнения которой начинается формирование отрицательного полупериода.
Обращаю Ваше внимание на следующий общий принцип: ***то, что рабочая точка программы "стоИт" на команде, означает, что исполнилась предыдущая команда, а команда, на которой "стоИт" рабочая точка программы, еще не исполнена.***
Проще говоря, команда **bcf PortB,2** исполнится только тогда, когда рабочая точка программы "прыгнет" на следующую, после нее, команду.
Рассуждаем дальше: начальная точка замера имеется, а конечной точки замера нет.
Вывод: значит, ее нужно назначить (диалектика …).
Ей должна быть команда, после исполнения которой, **0** меняется на **1** (**bsf PortB,2**).
Правой кнопкой мыши, щелкаем по строке с этой командой и выбираем ***Break***.
Итак, рабочая точка программы "стоИт" в начале отрицательного полупериода (на "верхней" точке остановки), а "нижняя" точка остановки установлена в начале положительного полупериода:



А теперь все просто: в окне секундомера, щелкаем по кнопке ***Zero*** (сбрасываем показания секундомера в ноль) и далее, по кнопке с **зеленым** светофором.
Дожидаемся установки рабочей точки программы на назначенную ранее точку остановки **bsf PortB,2** и снимаем показания секундомера:



В строке ***Time*** секундомера, Вы увидите величину интервала времени формирования **отрицательного** полупериода. Если Вы сделаете этот замер в отработанном тексте программы **cus**, то увидите, что время формирования отрицательного полупериода точно равно расчетному (**345 мкс**., см. "вышележащую" картинку).
Для того чтобы замерить величину интервала времени формирования **положительного** полупериода, нужно, не снимая точек остановок, просто обнулить секундомер, а затем еще раз щелкнуть по **зеленому** светофору и дождаться результата замера.
Но, как говорится, "легких путей нам не нужно" (в обучающих и тренировочных целях), и поэтому еще раз проделаем то же самое, но со снятием точек остановок (***Debug → Clear All Points***).
Устанавливаем точку остановки на команде **bsf PortB,2**
В "автомате", переходим на эту точку остановки.
Устанавливаем точку остановки на команде **bcf PortB,2**
Сбрасываем секундомер на ноль (***Zero***).
Щелкаем по кнопке с **зеленым** светофором и ждем окончания процесса:



Смотрим на показания секундомера.
Величина интервала времени формирования положительного полупериода равна **344 мкс.** (замер производится в окончательно отработанном тексте программы).
Почему не 345 ?
А для разнообразия. В обучающе-тренировочных целях (для стимуляции мыслительной деятельности).
Если требуется выставить 345 мкс., то нужно либо добавить еще один **nop**, например, после команды **bsf PortB,2** (или перед командой **movlw .83**), либо вообще убрать все три **NOP**а и увеличить числовое значение константы с **.83** до **.84** (напоминаю, что один цикла ПП **PAUSE_2** отрабатывается за **4 мкс.**).
Итак, на отлаженной программе, я показал Вам, как **проверить** числовые значения интервалов времени формирования отрицательного (нулевой уровень) и положительного (единичный уровень) полупериодов.
Отлаживать тут нечего, так как программа была ранее отлажена.
Теперь нужно понять, что такое **отладка программы**.
Предположим, что мы работаем с текстом программы, написанной "с чистого листа".
Чтобы перейти к такому тексту, нужно несколько видоизменить текст программы **cus**.
Вносим, оговоренные выше, изменения.
Уберите все **NOP**ы (6 штук), которые находятся после команд **bcf PortB,2** и **bsf PortB,2** (можно или поставить перед ними точки с запятыми, или вообще убрать их из текста программы: первое → удобнее, а последнее → нагляднее).
В группах команд записи констант, замените **.85** на **.86** и **.83** на **.86** (напоминаю, что **.86** это рассчитанное ранее значение).
Изменения в текст программы внесены, следовательно, его нужно проассемблировать.
Делаем это и убеждаемся, что ошибок нет.
Итак, в обучающих целях, "забываем" про отлаженный текст программы, с которым работали ранее (что-то типа запланированного "склероза").
Перед нами → текст программы "с чистого листа".
То есть, текст, в котором калибровочные константы, задающие значения интервалов времени формирования отрицательного и положительного полупериодов установлены "на глазок" (не точно).
Задача: необходимо установить интервалы времени формирования отрицательного полупериода равным 345 мкс., а положительного - 344 мкс.
Все начинается с проверки фактических величин этих интервалов времени.
Это будут как бы "начальные точки отсчета".
По ходу исполнения программы, первым формируется отрицательный полупериод, значит с него и нужно начать.
Предполагаю, что Вы изучили изложенную выше информацию и потренировались в установке и снятии точек остановки, а также и разобрались с секундомером.
После сброса программы на начало, "дойдите" до команды **bcf PortB,2** либо в режиме пошагового исполнения программы, либо в "автомате" (описано выше, и про "уловку" не забудьте). Лучше в "автомате" (быстрее).
Итак, рабочая точка программы находится на команде **bcf PortB,2** (начало формирования отрицательного полупериода).
Вызываем секундомер и сбрасываем его на ноль.
"Начальная точка отсчета" имеется.
Назначаем точкой остановки команду **bsf PotrB,2** (начало формирования положительного полупериода).
Щелкаем по кнопке с **зеленым** светофором и ждем окончания процесса подсчета.
Смотрим на показания секундомера. Они равны **346 мкс**. "Перебор".
Значит, значение константы нужно уменьшить.
Если уменьшить ее на 1, то есть, сделать равной **.85**, то время формирования отрицательного полупериода уменьшится на **4 мкс.** и составит **342 мкс.**
"Недобор" в 3 мкс. устраняем добавлением трех **NOP**ов, либо сразу же после команды **bcf PortB,2**, либо сразу же после команды **goto PAUSE_1**.
Можно также "врезать" эти **NOP**ы после команды **movwf Sec** или между командами **movlw .85** и **movwf Sec**.
А теперь "вспомните" (склероз выключен) отстроенный текст программы.

Вот Вам и ответ на вопрос: откуда что взялось?
В данном случае, я "врезал" **NOP**ы после команды **bcf PortB,2** (на мой взгляд, так симпатичнее).
Получилось то, что нужно: **345 мкс.** (ранее проверено по секундомеру).
Измеряем время положительного полупериода.
Сбрасывать программу на начало и снимать точки остановки не нужно.
Примечание: если Вы "добирались" до команды **bcf PortB,2** в режиме пошагового исполнения программы (вспомните, что в этом случае, точку остановки устанавливать не обязательно), то сделайте ее точкой остановки.
Если Вы "добирались" до команды **bcf PortB,2** в "автомате", то она уже является точкой остановки (а иначе, в "автомате", Вы бы до нее не "добрались") и делать ничего не нужно.
Сбрасываем секундомер на ноль.
Щелкаем по кнопке с **зеленым** светофором и ждем окончания подсчета.
Получаем **353 мкс.**
А нужно **344 мкс.** "Перебор" в 9 мкс.
Если уменьшить величину константы на 2 (минус 4х2=8 мкс.), то опять будет "перебор".
Следовательно, необходимо уменьшить величину константы на 3, а "недобор" скомпенсировать добавлением трех **NOP**ов, что Вы и видите в тексте отлаженной программы.
Из этого следует общий принцип: ***если существует "перебор" и нет возможности скомпенсировать его удалением, из текста программы, соответствующего количества команд*** (бывают случаи, когда это возможно)***, то числовое значение константы изменяется таким образом, чтобы возник минимальный "недобор", который, в дальнейшем, компенсируется добавлением соответствующего количества NOPов.***
Все, процесс отладки программы **cus** закончен.
Точки остановки можно снять.
Теперь нужно убрать "обманный" **nop** и "вернуть на свое бывшее место" команду **goto PRD**.
Остается только проассемблировать текст программы (так как были внесены изменения), и после этого, как говорится, "со спокойной совестью", можно открыть, созданный при этом ассемблировании, **HEX-файл** программы, в программе, обслуживающей Ваш программатор, и произвести "прошивку" ПИКа.
Описанный ранее, процесс предварительного, "грубого" расчета значений времязадающих констант и последующей отладки программы, достаточно прост, но не все так просто.
А если ПП задержки имеет в своем составе "мощную врезку", и даже "грубый" расчет времени отработки такой задержки вызывает существенные затруднения?
В этом случае, можно просто "утонуть в расчетах", потратив на это уйму времени и сил.
Способ преодоления такого рода затруднений, как это не странно, достаточно прост.
Например, имеется 3-хразрядный счетчик с "массивной врезкой", и у Вас, в связи с этим, возникли затруднения с предварительными расчетами числовых значений констант или Вам просто не хочется ими заниматься.
Что нужно делать в этом случае?
В старшем и среднем разрядах счетчика выставляем значения констант, из расчета формирования малого интервала времени.
Если разряд счетчика → вычитающий, то выставляется константа, например, **.02**, а если суммирующий, то, например, **.254**.
Это делается для удобства. Чтобы в секундомере, подсчет времени происходил быстро и не нужно было бы долго ждать его конца.
Для младшего разряда счетчика (он "шустёр), выставляем константу "от балды", но не максимальную.
Например, **.10** ("привязка к круглой цифре". Для удобства).
"Выходим" на начало отсчета, назначаем "нижнюю" точку остановки, сбрасываем секундомер в ноль и запускаем "автомат".
Ждем окончания подсчета и смотрим на показания секундомера.
Например, получилось **1250 мкс.**
В младшем разряде счетчика, увеличиваем значение константы с **.10** до **.11** .
Делаем те же "манипуляции".
Получилось, например, **1353 мкс.**
Вычисляем разницу: 1373-1250=**123 мкс.**

Вывод: **"шаг" изменения константы младшего разряда** составляет **123 мкс.**
Следовательно, полный цикл счета счетчика младшего разряда составляет 256x123=**31488 мкс.**
Это означает, что через каждые **31488 мкс.**, содержимое счетчика среднего разряда будет изменяться на единицу.
Следовательно, полный цикл счета счетчика среднего разряда составит 256x31488=**8,060928 сек.**
Это означает, что через каждые **8,060928 сек.**, содержимое счетчика старшего разряда будет изменяться на единицу.
Следовательно, полный цикл счета счетчика старшего разряда составит 256x8,060928=**2063,5975 сек.**
Это есть максимальное время задержки ("грубая прикидка").
Например, требуется сделать "грубую прикидку" значений констант для вычитающего, трехразрядного счетчика, формирующего интервал времени 20 сек.
В старшем разряде, тройку назначать нельзя, так как будет "перебор", а двойка будет в самый раз: 2x8,060928=**16,121856 сек.**
Остаток: 20-16,121856=**3,878144 сек.**
Руководствуясь таким же принципом, для среднего разряда счетчика, назначаем константу **.123**.
123x31488=3873024 мкс.=**3,873024 сек.**
Остаток: 3,878144 сек.- 3,873024 сек.=0,00512 сек.=**5120 мкс.**
Для младшего разряда счетчика, назначаем константу **.41** (с "недобором"): 41x123=**5043 мкс.**
То, что сейчас проделано, называется **разложением**.
Итак, "грубая прикидка" значений констант, для формирования интервала времени величиной 20 сек., дала следующие результаты:
- для старшего разряда: **.02** (H),
- для среднего разряда: **.123** (M),
- для младшего разряда: **.41** (L).

Обращаю Ваше внимание на то, что это именно "грубая прикидка".
И не только по той причине, что в младшем разряде, константа назначена приблизительно, но и по причине того, что в состав "куска" программы, во время отработки которого формируется калиброванный интервал времени, могут входить команды, которые не имеют непосредственного отношения к подпрограмме задержки.
То есть, речь идет о сумме фактического времени отработки задержки и времени отработки этих команд.
Именно поэтому, в младшем разряде должен быть "недобор".
Вернемся к рассмотренному выше примеру: при значении константы младшего разряда счетчика **.10** и шаге в **123 мкс**., счетчик должен показать **1230 мкс.**, но он показывает **1250 мкс.** (это пример того, что часто встречается).
В чем дело?
Дело в том, что кроме ПП задержки, еще отрабатывается (в течение 20 мкс.) и группа команд, которая не входит в ее полный цикл.
Такие "линейные вкрапления" могут быть "разбросаны" в различных местах "куска" программы, во время отработки которого формируется калиброванный интервал времени.
Утешает то, что в большинстве случаев, суммарное время отработки этих команд, по сравнению с временем отработки ПП задержки, мало.
Это позволяет достаточно точно "выйти в числовой сектор обстрела".
В нашем случае, если имеются такие "линейные вкрапления", реальная величина калиброванного интервала времени может оказаться более чем 20 сек.
А может и не оказаться.
В любом случае, при помощи секундомера, можно соответствующим образом скорректировать числовые значения констант, начиная с младшего разряда, и максимально приблизиться к желаемому, используя **NOP**ы.
Во многих случаях, достаточно только скорректировать числовое значение константы младшего разряда.
В случае значительных расхождений между фактической и расчетной величинами, нужно замерить "шаги" констант среднего и/или старшего разряда и внести в расчеты соответствующие коррективы.
Просто нужна "тренировка".

Когда Вы "набьете на этом деле руку", все это будет происходить без "напряга", хотя и будет занимать определенное время.
Главное - понимать смысл производимых действий. Ну и "технологию" тоже.
Потренироваться можно, используя все ту же программу **cus**.
Если у Вас есть желание, то попробуйте изменить величину трехсекундного интервала времени "выхода" сигнала тонального вызова в эфир, например, до 0,500000 сек. (или меньше) и максимально приблизиться к этому значению.
Большее время задавать не нужно (будете долго ждать конца счета).
Попробуйте сами назначить точки остановки.
Не забудьте про "уловку".
Разбираем следующий случай: **в тексте программы имеется ошибка.**
Если это ошибка не функционального характера, то **MPLAB** обязательно на нее укажет.
Например, Вы забыли "прописать" регистр, а в тексте программы есть обращение к нему или допустили орфографическую ошибку при написании команды, или забыли указать место сохранения результата операции, или забыли указать номер бита, с которым производится действие, или орфография названия регистра, в рабочей части программы, не соответствует орфографии "прописанного" названия этого же регистра, или один столбец "наезжает" на другой, или Вы написали строку с комментариями и не поставили перед ней точку с запятой и т.д.
В этом случае, после ассемблировании текста программы, Вы получите сообщения о всех допущенных ошибках, с указанием строк программы, в которых они допущены.
Двойной щелчок по строке с ошибкой, и курсор покажет Вам ту строку текста программы, в которой она допущена.
Эти ошибки нужно устранить и добиться безошибочного ассемблирования.
При этом, вовсе не обязательно, за один "присест", устранять все ошибки (если их много).
Можно, не спеша и по очереди, разобраться с каждым сообщением об ошибке, производя ассемблирование после каждой такой "разборки" или после неудачной ее попытки, вплоть до устранения ошибки.
Пока все ошибки не будут устранены, дальнейшая работа с текстом программы просто не имеет смысла (**MPLAB** "уйдет в отказ" и о создании HEX файла можно даже и не мечтать).
Советую Вам попробовать внести в текст программы **cus** какие-нибудь ошибки, из списка тех, что указаны выше, и посмотреть, что из этого получится после ассемблирования.
Предположим, что все ошибки подобного рода устранены и получено сообщение о безошибочном ассемблировании.
После этого, "хлопать в ладоши", вообще-то, рановато, так как далее предстоит "борьба" с ошибками функционального характера (естественно, при их наличии), что будет посерьезнее, чем ошибки в оформлении текста программы.
По той причине, что в этом случае, **MPLAB** не поможет (ошибок функционального характера "он не видит").
Ошибки функционального характера возникают из-за непродуманности как блок-схемы в целом, так и способов реализации ее "квадратиков" и/или связей между ними.
Например, рабочая точка программы "зависла" в каком-то цикле и "не хочет из него выходить" даже при соблюдении всех условий, обеспечивающих такой выход, или выполнение какой-то ПП игнорируется, не смотря на то, что она, по замыслу, должна исполняться и т.д.
Такого рода ошибок может быть великое множество и "диапазон их причин простирается" от элементарной невнимательности, когда можно обойтись "малой кровью", и до ошибок стратегического характера, когда нужно производить "капитальный ремонт" всей программы.
Суть поиска функциональных ошибок сводится к проверке исполнения программы (в симуляторе и/или в "железе"), с контролем соответствия фактических результатов, желаемым результатам.
В случае отсутствии такого соответствия, выявляется причина несоответствия и принимаются меры для его устранения.

Анализ текста программы на наличие или отсутствие ошибок функционального характера целесообразно начать с "прогонки" рабочей точки по всему полному циклу программы.
В этом случае, работа происходит по принципу "следопыта".
Первичная задача: нужно "протащить" рабочую точку программы по полному циклу программы, начиная с первой ее команды, при настройках **MPLAB** по умолчанию.

До возникновения первого затруднения.
Далее → работа мозга и радость победы (в идеале).
После устранения первого затруднения, рабочая точка программы "протаскивается" дальше, до возникновения второго затруднения.
И так далее. До конца полного цикла программы (вернее, до "выхода на новый виток" полного цикла программы).
Факт такого "протаскивания" рабочей точки программы по полному циклу односценарной программы, соответствует проверке работоспособности программы типа "грубо" и говорит о том, что программа "жизнеспособна".
После этого, можно приступить к более детальной проверке указанного выше соответствия.
После того, как программист убеждается в соответствии фактического алгоритма работы программы расчетному (задуманному им), можно перейти к отладке программы (если она требуется).
Если программа имеет несколько сценариев работы (многосценарная программа), то нужно "пройтись" по их максимально возможному количеству (в идеале, по всем ее сценариям).
Работу подавляющего их большинства можно проверить, либо используя "уловки", либо используя функции стимула, но встречаются и такие сценарии работы программы, которые, из-за сложности или невозможности имитации внешних воздействий, сложно или вообще невозможно проверить в симуляторе (надежда только на мозги и смекалку. Тренируйте их!).
В этом случае, после каждого изменения текста программы, необходимо ее ассемблировать, "зашивать" в реальный ПИК и смотреть, что из этого получится.
Есть такие "штуковины" под названием эмуляторы, но как глянешь на их цены, становится очень грустно. Лучше представить себе, что их нет и полагаться на свое мастерство.

Обращаю Ваше внимание на то, что употребленное мной слово "затруднение", вовсе не свидетельствует о наличии ошибки функционального характера.
Затруднение может быть связано с ней, но оно может быть также связано и с "уходом" рабочей точки программы в нежелательный сценарий работы программы.
Вспомните про то, как при отладке программы **cus**, рабочая точка программы "ушла в вечное кольцо" ПП **PRD**, и из-за этого нельзя было осуществить замер значений полупериодов.
В этом случае, никакой функциональной ошибки не было. Просто внешнее воздействие было таким, что исполнение программы происходило по нежелательному сценарию.
Это вовсе не говорит о том, что этот нежелательный сценарий нежелателен вообще.
Проверить нужно и его, но только в ходе "прогонки", осуществляемой именно для проверки этого сценария.
В зависимости от сложности программы (от количества сценариев ее исполнения), таких "прогонок" может быть достаточно большое количество, так что освоение их "технологии" (свободная "рулёжка" сценариями программы) → насущная необходимость.
Чем больше способов имитации внешних, управляющих сигналов знает программист, тем в меньшее количество "тупиков" он будет попадать.
Вот Вам и ответ на вопрос: "Зачем нужны функции стимула и/или уловки"?
На мой взгляд, эффективнее сконцентрировать внимание не на функциях стимула, а на "уловках".
Лично я, так и делаю.
Далее, по мере "подворачивания поводов", я буду "выдавать на гора" другие разновидности "уловок".
В следующем разделе, тема "рулёжки" сценариями будет продолжена.

10. Как отследить выполнение программы

Важность этой работы очевидна.
"Набор" приемов отслеживания выполнения программы вовсе не является "резиновым".
"Овладение" этими приемами, хотя и требует определенных усилий, но вовсе не является чем-то очень уж сложным.
Сознательно "манипулируя" этими приемами, можно составить четкое представление о том, исполняется ли программа в соответствии с задуманным алгоритмом ее работы или "где-то что-то пошло наперекосяк", и по какой причине этот "перекосяк" произошел.
То есть, речь идет о сборе данных для последующего анализа причин "перекосяка", с целью его устранения.
В ходе работ по отслеживанию выполнения программы, выявляются ошибки функционального характера, а по результатам их анализа, "по ним заказывается панихида".
Давайте с этим разбираться.
Отследить перемещение рабочей точки программы, по тексту программы, можно ***в текстовом редакторе*** **MPLAB** или в окне ***Program Memory Window*** (вызывается кнопкой **ROM**).
В этом окне, все команды программы располагаются компактно, с указанием их порядковых номеров и адресов в памяти программ, но названий регистров Вы там не найдете (вместо них указаны их адреса в области оперативной памяти), все числа → только в 16-ричной форме и комментариев нет.
В большинстве случаев, отслеживание движения рабочей точки производится в текстовом редакторе **MPLAB**, так как это гораздо более комфортно, а окно ***Program Memory Window*** "вызывается" тогда, когда нужно узнать порядковый номер той или иной команды или ее адрес в памяти программ, а также разложение (на команды) директив/макросов.
В окне ***Program Memory Window*** можно назначить точки остановки.
Технология этого назначения такая же, как и при назначении точек остановки в текстовом редакторе: щелчок правой кнопкой мыши по строке с командой и выбор (щелчок левой кнопкой мыши) из списка строки ***Break***.
После этого, выбранная строка (в окне ***Program Memory Window***) окрашивается в **красный** цвет, и в текстовом редакторе, Вы увидите то же самое (как будто бы Вы назначили точку остановки в текстовом редакторе).
Окно ***Special Function Register Window*** (кнопка **SFR**).
Это окно бывает полезным при работе с программами, которые управляются внешними сигналами.
Если Вы заранее не предприняли мер по установке необходимых Вам уровней этих внешних, управляющих сигналов (функции стимула не задействованы), то **MPLAB**, на момент начала исполнения программы, "тупо" выставит на этих выводах "свои" уровни (по умолчанию).
Откройте программу **cus** (сбрасывать ее на начало и исполнять не нужно, просто откройте и все).
Щелкните по кнопке **SFR**.
Откроется окно ***Special Function Register Window***, в котором Вы увидите эту настройку по умолчанию.
В основном, в регистры специального назначения записаны нули, за исключением 4-х регистров, строки которых окрашены в **красный** цвет.
Во всех битах регистра **TrisB** записаны единицы, следовательно, на момент начала исполнения программы, все выводы порта B "виртуального" ПИКа работают на вход.
Если эти состояния не изменить программными средствами (имеется ввиду и программа, с которой происходит работа, и **MPLAB**), то они такими и останутся.
То же самое относится и к регистру **TrisA**.
С поправкой на то, что в трех его старших битах, по умолчанию, выставлены нули.
В регистре **Option** выставлены все единицы.
Чем руководствовались при этом разработчики **MPLAB**, остается только гадать.
Это просто нужно "принять как данность".
В регистре **Status**, биты №3 и 4 установлены в 1, что соответствует, якобы, прохождению сброса по включению питания ("якобы" потому, что ПИК "виртуальный").
Если сбросить программу на начало, то в окне ***Special Function Register Window*** ничего не изменится, за исключением того, что **красное** выделение пропадет.

Из оставшихся настроек "по умолчанию", прежде всего, необходимо обратить внимание на нули в регистрах **PortA** и **PortB**.
Ранее мы уже сталкивались с тем, что из-за этого возникали затруднения (переход в "ненужный" сценарий работы программы) при отладке программы **cus**.
По этой же причине, они могут возникать и при отладках других программ.

Вывод из этого простой: ***если в результате исполнении команд, обращающихся к регистрам PortA или PortB, возникнут затруднения с переходом в нужный сценарий работы программы, то необходимо открыть окно Srecial Function Register Window, посмотреть на содержимое этих регистров (на момент исполнения команды) и сделать соответствующие выводы.***

Что касается остальных регистров, то нужно учесть, что если к их содержимому, по ходу исполнения программы, не было обращений, то в них так и останутся нули (за исключением флагов).
Примечание: это не относится к регистру **PCL**, так как его содержимое, по ходу исполнения программы, постоянно изменяется.
По ходу исполнения программы, содержимое задействованных в программе регистров специального назначения, естественно, будет изменяться.
Таким образом, можно отследить все изменения, происходящие с их содержимым.
При этом, строки, в которых находятся регистры, содержимое которых, в результате исполнения текущей команды, изменилось, выделяются **красным** цветом, что удобно с точки зрения акцентирования внимания на произошедших изменениях.
Так как содержимое регистров специального назначения удобнее всего воспринимать в бинарном виде, то окно ***Special Function Register Window*** может оказаться весьма полезным.
Но основным, "рабочим" окном является окно ***File Register Window*** (кнопка **RAM**).
Оно удобно тем, что в нем можно "оптом" отследить все изменения, происходящие как с содержимым регистров специального назначения, так и с содержимым регистров общего назначения (контролируется вся область оперативной памяти).
Если Вы сравните содержимое регистров специального назначения, расположенных в этом окне, с содержимым регистров специального назначения, расположенных в окне ***Special Function Register Window*** (с учетом перевода чисел из одной системы исчисления в другую), то обнаружите полное соответствие, причем, в интервале времени от открытия проекта и до сброса программы на начало, содержимое указанных выше, 4-х регистров специального назначения, будет также выделено **красным** цветом (после сброса программы на начало, **красное** выделение снимается).
Это полное соответствие наблюдается и по ходу исполнения программы.
Обычно, в течение основной части времени отслеживания работы программы, пользуются окном ***File Register Window*** (оно более информативно), а окно ***Special Function Register Window*** открывают по мере необходимости (а можно открыть оба, и не закрывать их до конца работы).
Неудобство применения окна ***File Register Window*** связано с тем, что названия регистров специального назначения в нем не указываются (указываются адреса), и это, на первых порах, может вызвать затруднения в ориентации, что совсем не "смертельно".
Держите перед глазами распечатку области оперативной памяти, и со временем, Вы просто запомните "где что лежит" и в дальнейшем, сможете обходиться без нее.
Что касается названий и адресов регистров общего назначения (их присваивает программист), то для каждой программы они различны.
В распечатке области оперативной памяти, эти названия можно вписать в соответствующие прямоугольники, и при возникновении необходимости в "привязке" содержимого регистров, к названиям регистров, просто свериться с этой распечаткой.
По ходу исполнения программы, содержимое таблицы окна ***File Register Window***, в части касающейся задействованных в программе регистров, будет меняться.
Если происходит изменение содержимого любого из этих регистров, то результат этого изменения выделяется **красным** цветом.
Если после исполнения следующей команды, изменения этого содержимого не происходит, то **красный** цвет меняется на **синий**.

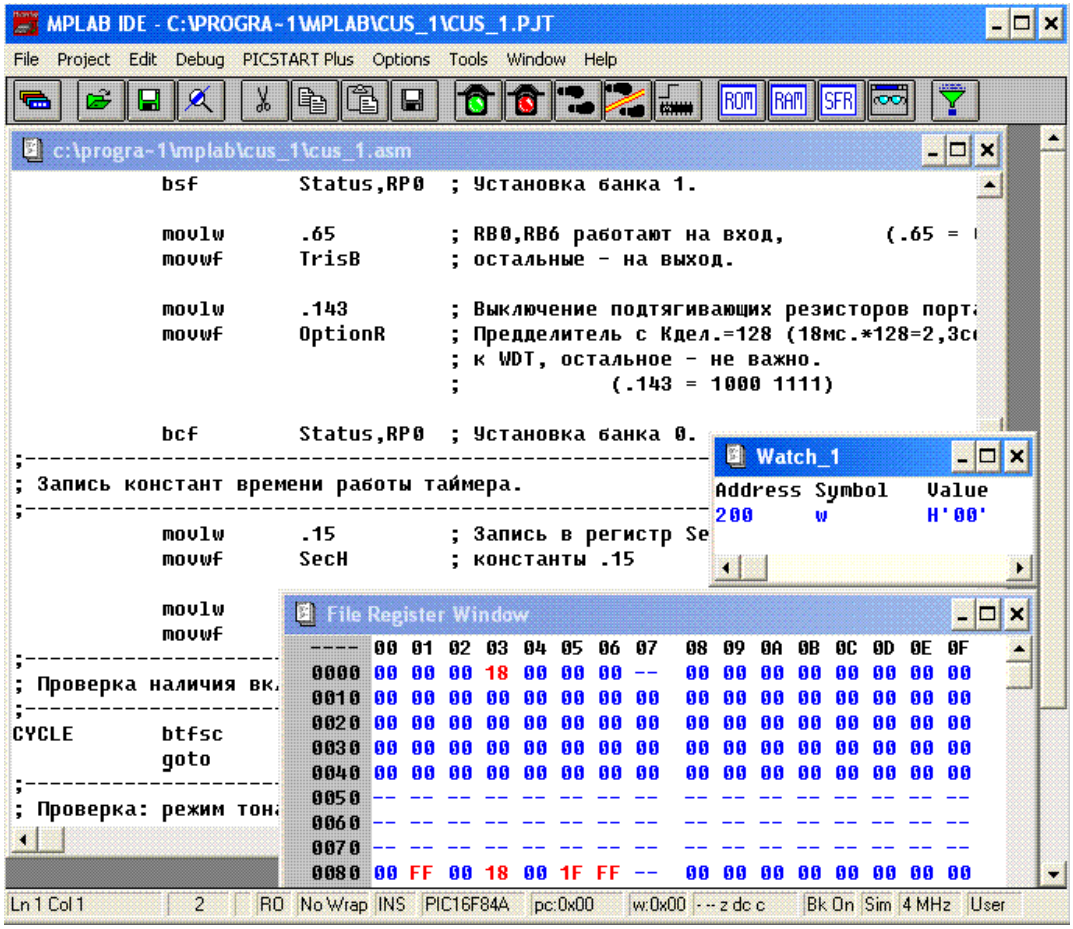
Таким образом, можно комфортно отследить изменения, происходящие в области оперативной памяти и их соответствие задуманному.
Так как в **PIC16F84A**, содержимое регистров общего назначения нулевого банка дублируется (отображается) в первом банке, то на последние 4 строки таблицы окна ***File Register Window*** можно не обращать внимания, а оставить только 1-ю строку 1-го банка (с регистрами специального назначения).
Лично я, изменяю размер этого окна таким образом, чтобы нижняя граница окна **MPLAB** "наехала" на эти 4 строки окна ***File Register Window*** и закрыла их.
Так оно занимает меньше места.
Что касается кнопки с очками, то "жать" на нее имеет смысл только в том случае, если необходимо отследить содержимое регистра **W** (в других окнах, содержимое регистра **W** не отслеживается).
В этом окне, конечно же, можно отследить содержимое и всех остальных регистров, но удобнее это сделать в окне ***File Register Window***.
К помощи этой кнопки можно обратиться в случае, если необходимо сохранить содержимое выбранных из списка регистров (в тот или иной момент исполнения программы), в виде отдельного файла.
Может быть, для кого-то это и окажется полезным, но лично у меня, никогда не возникало такой потребности.
Если возникает потребность в отслеживании содержимого регистра **W**, то он заносится в окно ***Watch*** (как это делается, описывалось ранее).
Если предельно уменьшить размеры этого окна, то оно получается очень компактным и "малогабаритным".
По этой причине, его можно один раз открыть и не закрывать до конца работы.

А теперь, на практике, отследим выполнение "учебно-тренировочной" программы cus.

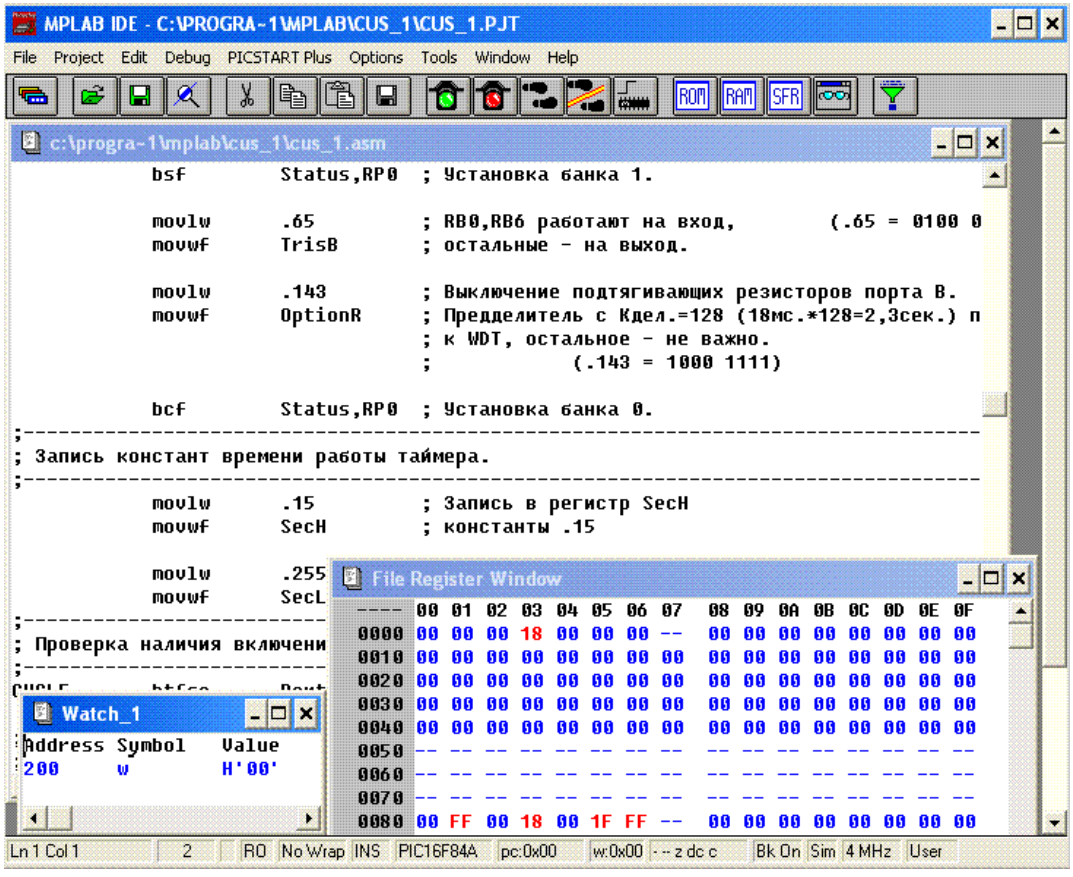
Откройте любой из проектов **cus_1** или **cus_2**.
Сначала будем работать с отлаженной программой (команда **goto PRD** на **nop** не заменялась).
Примечание: для краткости, в дальнейшем, я буду заменять названия окон названиями кнопок, при помощи которых они открываются:

- Program Memory Window на ROM,
- File Register Window на RAM,
- Special Function Register Window на SFR,
- Watch - без изменения.

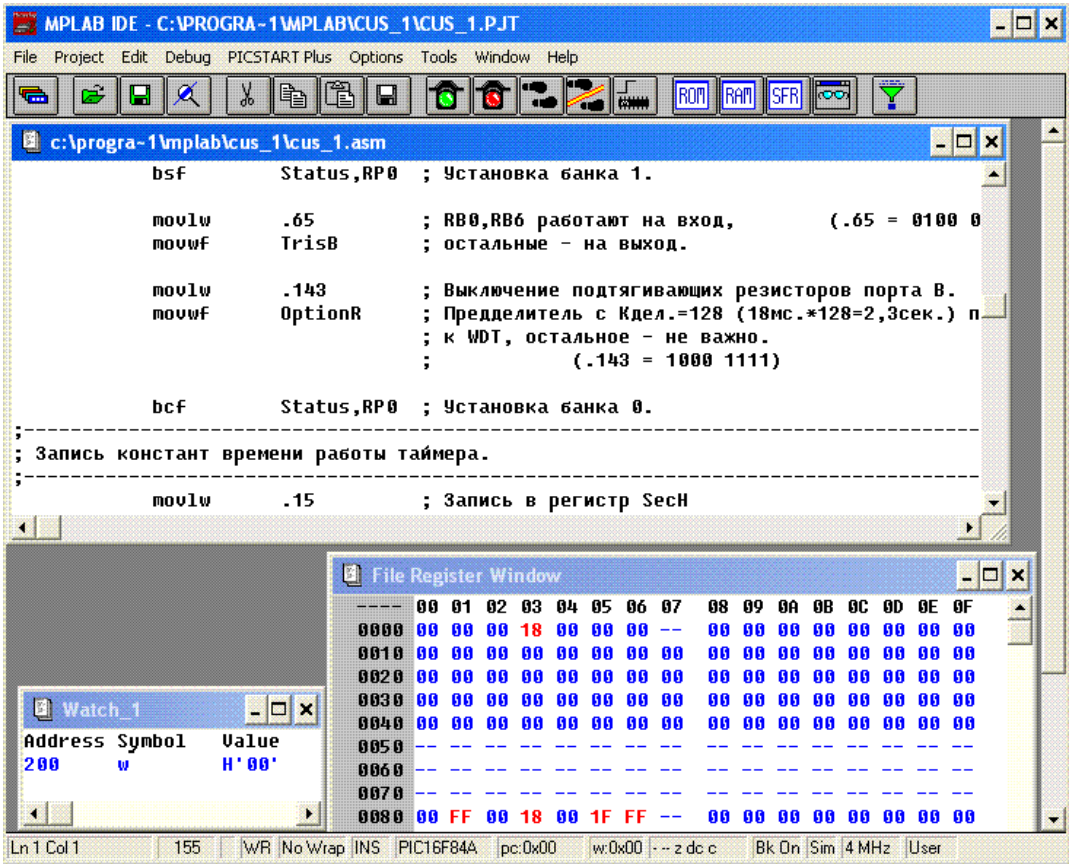
"Наводим порядок на рабочем месте".
Если "лист" текстового редактора раскрыт полностью, то уменьшите его размеры, отодвинув к центру правый и нижний срезы этого "листа" на 1 - 2 см.
В этом случае, всегда будут видны "краешки" других открываемых нами окон (если они не активны и находятся за "листом" текстового редактора) и проблем с их "активацией" не будет (просто так удобнее).
Открываем окно **RAM** и перемещаем его в правый нижний угол окна **MPLAB**.
"Цепляемся" за верхний срез окна **RAM** и двигаем его вниз до тех пор, пока нижние 4 строки таблицы перестанут быть видны.
Создаем окно ***Watch***, "заложив" в него регистр **W**.
Размещаем это окно выше окна **RAM** (нижний срез окна ***Watch*** - "впритык" к верхнему срезу окна **RAM**), и правый срез окна ***Watch*** "двигаем вправо до упора".
"Цепляемся" за верхний левый угол окна ***Watch*** и уменьшаем его размер до минимально возможного.
Получилось так:



А можно и так:



Или так:



"Прожиточный минимум" выставлен.
Остальные окна можно открывать по мере необходимости.
Активация окон происходит обычным для Виндов способом, то есть, щелчком в любом месте окна.
Естественно, что детально отследить работу программы можно только при пошаговом ее исполнении.
Поэтому щелкаем по кнопке со следами или жмем **F7** (сброс программы на начало).
Для краткости, "привяжемся" к кнопке **F7** клавиатуры.
Итак, жмем на **F7** и переходим на первую команду ПП **START**.
То, что рабочая точка программы "встала" на команду **clrf IntCon**, означает то, что исполнилась не эта команда, а предшествующая, то есть, команда **goto START** (та, что в "шапке" программы).
Смотрим в окна **RAM** и ***Watch***.
Изменилось только содержимое регистра с адресом **02h** (содержимое выделилось **красным** цветом).
Смотрим в распечатку области оперативной памяти: по этому адресу расположен счетчик команд (регистр **PCL**), и его содержимое изменилось с **00h** на **01h**, то есть, произошел переход рабочей точки программы на вторую команду программы, а предшествующая (первая) команда была исполнена.
По причине того, что по ходу исполнения программы, содержимое регистра **PCL** последовательно изменяется, я не буду "заострять" Ваше внимание на его содержимом.
Регистр **PCL** задействуется всегда (при выполнении любой программы), и его содержимое всегда окрашено в **красный** цвет (сами ответьте на вопрос: почему?).
Итак, исполнилась команда **goto START**.
Смотрим в распечатку команд: команда **goto** выполняется за 2 м.ц. (можете вызвать секундомер и проверить) и при этом, никаких операций с содержимым регистров не производится (эта команда не меняет их содержимого и на состояния флагов не влияет).
Поэтому не удивительно, что кроме изменения показаний счетчика команд, никаких других изменений, в открытых нами окнах, не произошло.
Нажимая **F7**, выполняем команду **clrf IntCon**.
Рабочая точка программы "перескочила" на команду **clrwdt,** и содержимое регистра, с

адресом **03h,** окрасилось в **красный** цвет (изменение с **18h** на **1Ch**).
Смотрим в распечатку области оперативной памяти.
Этот адрес имеет регистр **Status**.
Смотрим в распечатку команд.
При исполнении команды **clrf** должен установиться флаг нулевого результата (**Z**), что и произошло.
Проверяем: **18h=00011000**, **1Ch=00011100**.
Все правильно: **2**-й бит (или бит **№ 2**) регистра **Status**, с названием **Z**, установился в **1**.
Вопрос: "Почему содержимое регистра **IntCon** (адрес **0Bh**) не окрасилось в **красный** цвет, ведь с его содержимым производилась операция?
Ответ: потому, что в результате инициализации "виртуального" ПИКа (по умолчанию), в этом регистре "выставлен" **0**, а при сбросе нуля в ноль получается ноль.
Это называется **подтверждением ранее установленного состояния**.
То есть, в конечном итоге, изменения содержимого регистра **IntCon** не произошло, а значит и выделения **красным** цветом не будет.
Нажимая на **F7**, выполняем команду **clrwdt**.
Ничего не изменилось, кроме показания счетчика команд.
Эта команда управляет аппаратной "начинкой" ПИКа, а если конкретно, то сторожевым таймером, и устанавливает (в единицы) биты (флаги) регистра **Status** с названиями **-TO** и **-PD** (см. распечатку команд).
Но эта установка уже была произведена ранее. По умолчанию.
То есть, и в этом случае, имеет место быть подтверждение ранее установленного состояния.
Выполняем команду (***далее, это словосочетание означает нажатие/отжатие F7***)
bsf Status,RP0.
Содержимое регистра **Status** изменилось с **1Ch (00011100) на 3Ch (00111100)**.
Все правильно: в **5**-м бите регистра **Status** (выбор банка) установилась **1** (выбор 1-го банка).
Выполняем команду **movlw .65**
Теперь **"покраснела"** строка в окне ***Watch***.
То есть, произошла запись числа **.65** в аккумулятор (в регистр **W**).
Переводим десятичное число **.65** в 16-ричное и получаем **41h**.
Сверяемся: это число и записалось (правда, буквы **h** нет, но она подразумевается).
Выполняем команду **movwf TrisB**.
"Покраснело" содержимое регистра **TrisB**, и в него, из регистра **W**, скопировалось (именно скопировалось, а не вырезалось → см. регистр **W**) число **41h**.
Переводим число **41h** в бинарную форму. Получается **01000001**.
Все правильно: 0-й и 6-й биты установлены в **1** (выводы **RB0** и **RB6** работают "на вход", а остальные, "на выход").
Следующие две команды выполняются аналогично, только число **.143** копируется, из регистра **W**, в регистр **OptionR**.
Убедитесь в правильности их выполнения самостоятельно.
Так же самостоятельно убедитесь в том, что после исполнения команды **bcf Status,RP0**, в **5**-м бите регистра **Status**, **1** сменилась на **0** (выбор 0-го банка).
С учетом этого, проверка исполнения следующих 4-х команд не должна вызвать у Вас затруднений.
Разница только в том, что константы (через **W**) записываются не в регистры специального назначения, а в регистры общего назначения (**SecH** и **SecL**), адреса которых (**0Dh** и **0Eh**) Вы найдете в "шапке" программы.
В ходе исполнения этой группы команд, Вы увидите, что содержимое регистра **TMR0** (адрес **01h**) изменится, и далее, по ходу исполнения программы, в нем будет что-то подсчитываться.
Хотя, в данном случае, то что он считает, "по барабану" (**TMR0** в программе не задействован), но с познавательной точки зрения, разобраться с этим стОит.
Настройки **TMR0** производятся с помощью битов регистра **OptionR**.
В него мы записали число **.143** (**10001111**).
5-й бит этого регистра установлен в **0**, следовательно, на вход **TMR0** подан внутренний тактовый сигнал (называется **CLKOUT**).
То есть, **TMR0** будет считать количество машинных циклов.
На работу программы **cus** это никоим образом не влияет, но если кому-то из Вас не нравится "лишняя точка отвлечения внимания", то от входа **TMR0**, можно отключить сигнал внутреннего такта и подключить вход **TMR0** к внешнему такту, то есть, к выводу **RA4/TOCKI**.

В данном случае, к выводу **RA4/TOCKI**, источник внешнего такта не подключен.
А раз это так, то в регистре **TMR0** зафиксируются нули, и его содержимое, по ходу исполнения программы, меняться не будет (можете проверить).
Для того чтобы это сделать, в команде **movlw .143**, нужно заменить константу **.143** на константу **.175**
Почему **.175** ? Ответьте на этот вопрос самостоятельно.
Если Вы внесли, в текст программы, это изменение, то естественно, что после этого необходимо произвести ассемблирование и снова исполнить программу так, как описано выше.
А еще лучше → произвести ассемблирование, после чего закрыть проект (с сохранением изменений), затем вновь открыть этот проект, и только после этого снова исполнить программу так, как описано выше.
Именно в последнем случае Вы увидите, в регистре **TMR0**, нули, а в первом случае, в нем может зафиксироваться число, отличное от нуля (результат предшествующего "гуляния" рабочей точки по программе).
Такого рода "хождения по мукам" полезны тем, что помогают понять следующее: ***настройкой "виртуального" ПИКа, по умолчанию, является настройка MPLAB на момент открытия проекта***, а не настройка по сбросу программы на начало, после того, как рабочая точка программы "погуляла" по программе.
Работаем дальше.
"Упираемся" в команду **btfsc PortB,0**.
Команда **btfsc** является командой ветвления, следовательно, нужно определиться, по какому из двух сценариев исполнения программы двигаться далее.
То есть, нужно посмотреть, в каком состоянии, перед исполнением этой команды, находится бит **№0** регистра **PortB**.
Для этого нужно открыть окно **SFR**, найти в нем регистр **PortB** и посмотреть на состояние этого бита.
В данном случае, по умолчанию, он установлен в **0** (функции стимула не используются).
После этого, окно **SFR** можно закрыть (а можно и оставить. Это кому как нравится).
Вывод: после исполнения команды **btfsc PortB,0**, "ухода в вечное кольцо" ПП **START** не произойдет.
То есть, после исполнения команды **btfsc PortB,0**, будет исполнена команда **btfss PortB,6**.
В ходе работы по отслеживанию правильности исполнения программы, программист может проверить как правильность исполнения этого сценария, так и правильность исполнения другого сценария.
В идеале, нужно "прогнать" рабочую точку программы по всем сценариям программы.
Предположим, что необходимо проверить правильность исполнения указанного выше сценария.
То есть, ноль, в нулевом бите регистра **PortB**, который **MPLAB** выставила по умолчанию, в данном случае, устраивает.
Выполняем команду **btfsc PortB,0**.
Рабочая точка программы переместилась на команду **btfss PortB,6**
Смотрим в окно **RAM**.
Изменений нет (кроме **PCL**), так как значение нулевого бита ***считывалось, а не записывалось.***
Перед выполнением команды ветвления **btfss PortB,6**, необходимо произвести проверку, аналогичную описанной выше.
После этого, выясняется следующее: с учетом текущего значения **6**-го бита регистра **PortB**, = **0** (выставлен **MPLAB** по умолчанию), после исполнения команды **btfss PortB,6**, будет исполнена команда **goto PRD** и рабочая точка программы "улетит" в ПП **PRD**.
Предположим, что отслеживание выполнения этого сценария предполагается отложить "на потом", а сейчас необходимо отследить выполнение другого сценария.
Для реализации этого, прибегаем к "уловке" (о ней говорилось ранее): меняем **goto PRD** на **nop**.
Так как в текст программы внесены изменения, то его нужно проассемблировать и выполнить программу, с самого начала и до команды **btfss PortB,6**.
Исполняем команду **btfss PortB,6**.
Смотрим в окно **RAM**: "картина" такая же, как и при выполнении команды **btfsc PortB,0** (и по той же причине).

Выполняем команду **nop**.
В окне **RAM** опять ничего не меняется (кроме **PCL**), так как эта команда не производит никаких действий.
Итак, рабочая точка программы "вышла на оперативный простор" и "стоит" на команде **bcf PortB,2**.
Исполняем эту команду.
В окне **RAM** опять ничего не изменилось (кроме **PCL**).
Почему, ведь команда **bcf PortB,2** является командой, влияющей на содержимое регистра?
Объяснение: по умолчанию, **2**-й бит регистра **PortB** установлен в **0**.
То есть, имеет место быть подтверждение ранее установленного состояния (**"краснеть"** не от чего).
Напоминаю, что проверку состояний битов регистров, лучше производить в окне **SFR**, так как в нем, содержимое регистров представлено в удобной для восприятия форме (бинарной).
Выполняем следующие 3 команды **nop**.
В окне **RAM**, по указанной выше причине, опять не произойдет никаких изменений (кроме **PCL**).
Рабочая точка программы "встала" на команду **movlw .85**.
"По образу и подобию", приведенного выше, отслеживания аналогичных команд (команд записи константы в регистр), отследите самостоятельно выполнение этой команды и следующей за ней, команды **movwf Sec**.
Убедитесь, что из регистра **W**, в регистр **Sec**, скопировалось число **.85** (**55h**).
Итак, рабочая точка программы установилась на первой команде (**clrwdt**) ПП задержки **PAUSE_1**.
Ранее, мы уже отслеживали выполнение этой команды, так что сделайте это самостоятельно.
Рабочая точка "встала" на команду **decfsz Sec,F**.
На момент ее исполнения, в регистре **Sec** (адрес **0Ch**) находится число **.85** (**55h**).
Посмотрите в окно **RAM** и убедитесь в этом.
Выполняем команду **decfsz Sec,F**.
Смотрим в окно **RAM**.
Изменение имеется: содержимое регистра **Sec**, в полном соответствии с алгоритмом выполненной команды, уменьшилось с **55h** до **54h**, то есть, на единицу (произошел декремент).
Так как число **54h** (результат операции, сохраненный в том же регистре **Sec**) отлично от нуля, то следующей исполняемой командой должна быть команда **goto PAUSE_1**.
Проверяем. И в самом деле, рабочая точка программы "стоИт" именно на этой команде.
Выполняем команду **goto PAUSE_1**.
В окне **RAM** никаких изменений (кроме **PCL**) не происходит (аналогично выполнению команды **goto START**).
Смотрим в текстовый редактор. Произошел безусловный переход рабочей точки программы на первую команду ПП **PAUSE_1** (**clrwdt**).
Снова выполняем команду **clrwdt**.
Снова выполняем команду **decfsz Sec,F**.
Смотрим в окно **RAM**. Содержимое регистра **Sec** еще раз уменьшилось на 1 (с **54h** до **53h**).
Число **53h** не равно **0**, значит далее, снова будет осуществлен безусловный переход в ПП **PAUSE_1** и так далее.
До тех пор, пока регистр **Sec** не очистится (**00h**).
Таким образом, рабочая точка программы "закольцевалась" в ПП **PAUSE_1**.
Если пытаться "добраться" до конца этой "закольцовки" (**00h** по адресу **0Ch**) пошагово, то это займет достаточно много времени (особенно, если счетчик - многобайтный).
Есть 2 способа уменьшения этого времени: либо назначить точкой остановки команду **bsf PortB,2** и "дойти" до нее в "автомате", либо поступить следующим образом: нажать на кнопку **F7** и не отпускать ее, одновременно контролируя содержимое регистра **Sec** в окне **RAM**.
Вы увидите, что содержимое этого регистра достаточно быстро начнет уменьшаться.
При приближении к нулю (например, где-то в районе чисел 01h - 04h), нужно остановить счет, отжав кнопку **F7**, и "добраться" до состояния **00h,** пошагово исполняя программу (с помощью всё той же кнопки **F7**).

В соответствии с алгоритмом работы команды **decfsz**, после исполнения команды последнего декремента (переход от **01h** к **00h**), рабочая точка программы перейдет на команду **bsf PortB,2**
При отслеживании работы программы, не нужно проверять соответствие фактических и требуемых значений временных параметров программы, так как оно проверяется при отладке программы.
При отслеживании работы программы, главное → "протащить" рабочую точку программы по выбранному сценарию работы программы (а в идеале, поочередно, по всем сценариям) до конца полного цикла программы, с одновременным контролем соответствия фактического и задуманного алгоритмов исполнения программы.
Обнаружение такого несоответствия, почти всегда, свидетельствует о наличии ошибки функционального характера, которую нужно исправить.
Для различных сценариев исполнения программы, полные циклы программы будут функционально различны (со всеми вытекающими последствиями).
Итак, рабочая точка программы находится на команде **bsf PortB,2**.
Так как группы команд формирования отрицательного и положительного полупериодов идентичны, то не буду повторять практически одно и то же.
С учетом сказанного выше, Вы вполне сможете самостоятельно отследить исполнение группы команд формирования положительного полупериода.
После формирования интервала времени положительного полупериода, рабочая точка программы установится на команде **decfsz SecL,F**.
Посмотрите в окно **RAM**.
На момент исполнения этой команды, регистр **Sec** очищен (в нем "лежит" **00h**), а в регистры **SecH** (адрес **0Dh**) и **SecL** (адрес **0Eh**) ранее записаны константы **.15** (**0Fh**) и **.255** (**FFh**) соответственно.
Выполняем команду **decfsz SecL,F**.
Смотрим в окно **RAM**.
Содержимое регистра младшего разряда **SecL** уменьшилось с **FFh** (**.255**) до **FEh** (**.254**), то есть на **1**, что соответствует алгоритму работы команды **decfsz** (что при условии сохранения результата операции в этом же регистре).
Результат выполнения операции (**FEh**) не равен нулю, следовательно, следующей исполняемой командой будет команда **goto CYCLE**.
Выполняем команду **goto CYCLE**.
После этого, рабочая точка программы "встаёт" на первую команду ПП **CYCLE** (**btfsc PortB,0**), что соответствует задуманному.
И в самом деле, после того как сформируется период сигнала тонального вызова, должно начаться формирование следующего периода, и т.д.
Вплоть до обнуления двухбайтного счетчика (то есть, до окончания трехсекундного интервала времени).
Таким образом, речь идет о длительной "закольцовке" рабочей точки программы.
Вопрос: "Как отследить такую протяженную по времени закольцовку"?
Ответ: в этом случае, проверка правильности исполнения алгоритма работы программы сводится к обнаружению факта очищения регистра старшего разряда счетчика, ведь именно после его очищения, имеет место быть сценарий типа "программа исполняется далее".
В данном случае, это единственный способ "выхода из закольцовки".
Так как речь идет о достаточно значительном интервале времени, то лучше всего произвести отслеживание факта "выхода из закольцовки" в "автомате".
Во всех остальных случаях (при пошаговом исполнении программы с кратковременными нажатиями **F7** и при пошаговом исполнении программы с долговременным нажатием на **F7**), такого рода отслеживание займет много времени.
Итак, убеждаемся, что декремент старшего разряда счетчика происходит (ранее мы убедились, что происходит декремент младшего разряда счетчика).
Для этого необходимо назначить точку остановки на следующей, после **decfsz SecH,F**, команде, то есть, на команде **goto CYCLE**.
Итак, назначаем точкой остановки команду **goto CYCLE**.
Сбрасываем программу на начало.
Для наблюдения за процессом счета, можно вызвать секундомер.
Щелкаем по кнопке с **зеленым** светофором ("автомат") и дожидаемся конца процесса счета

(придется подождать достаточно продолжительное время).
Через 175,71 мс. счет закончится и рабочая точка программы установится на команде **goto CYCLE**, следовательно, команда **decfsz SecH,F** "штатно" исполнена.
Смотрим в окно **RAM**.
Содержимое регистра **SecL** (младший разряд счетчика, адрес **0Eh**) очистилось (**00h**), а содержимое регистра **SecH** (старший разряд счетчика, адрес **0Dh**) уменьшилось на **1**, с **0Fh** (**.15**) до **0Eh** (**.14**), в чем и требовалось убедиться.
После этого, можно еще раз "запустить автомат": содержимое старшего разряда счетчика еще раз уменьшится на **1** (с **0Eh** до **0Dh**), а потом еще и еще..., пока счетчик старшего разряда не очистится.
Но эта очистка займет много времени, да и особой надобности в ней нет, так как для того чтобы убедиться, что 2-хразрядный счетчик работает "штатно", достаточно и пары декрементов содержимого счетчика старшего разряда.
Вопрос: "Каким образом "выйти" на сценарий "программа исполняется далее", не тратя массу времени на очистку содержимого регистра старшего разряда **SecH**"?
Ответ: нужно сделать то, что мы уже делали ранее → заменить следующую, после **decfsz SecH,F**, команду (**goto CYCLE**), на команду **nop**.
Делаем это.
В текст программы внесены изменения, следовательно, производим ассемблирование.
Теперь нужно удалить "старую" точку остановки и назначить "новую".
Но удалять старую точку остановки не нужно, так как она уже удалена при ассемблировании (***при ассемблировании, все точки остановки удаляются***), значит нужно только назначить новую точку остановки на команде **bcf PortB,2**.
Делаем это.
Запускаем "автомат" и дожидаемся его отработки.
После этого, рабочая точка программы установится на команде **bcf PortB,2**.
Таким образом, мы "коварно проскочили" 2-хразрядный счетчик и перешли в сценарий "программа исполняется далее", что и требовалось.
Если после этого, Вы посмотрите на показания секундомера и в окно **RAM**, то увидите ту же самую "картину", что и в случае одного декремента содержимого **SecH** : **175,71 мс.** и число **0Eh** по адресу **0Dh**.
То есть, в случае использования "уловки", **SecH** не очищается, и это естественно, так как при помощи одного декремента, **SecH** очистить нельзя.
Хотя и программа и не исполнена в полном объеме (**00h** в **SecH** нет), но тем не менее, мы можем, как говориться, с чистой совестью двигаться дальше, так как убедились, что счетчик работает по задуманному алгоритму, и получили гарантию "нормальной" его работы после замены "подставы" (**NOP**а) на "штатную" команду **goto CYCLE**.
Если у Вас есть время и терпение, то в познавательных целях, Вы можете добиться исполнения программы в полном объеме, многократно декрементируя содержимое регистра **SecH**, указанным выше способом.
В этом случае, **goto CYCLE** на **nop** менять не нужно.
Итак, мы находимся на команде **bcf PortB,2**.
Выполняем ее.
Смотрим в окно **RAM**.
2-й бит регистра **PortB** изменил свое состояние с **1** на **0** (содержимое PortB изменилось с **04h** на **00h**).
С учетом того, что перед исполнением команды **bcf PortB,2**, на выводе **RB2** присутствует единичный уровень, это соответствует задуманному.
Рабочая точка программы перешла на первую команду ("врезку") ПП **PRD** (**clrwdt**).
Выполняем команду **clrwdt**.
Рабочая точка программы установилась на команде ветвления **btfss PortB,0**
Следовательно, необходимо определиться, по какому из двух сценариев ее направить.
Предположим, необходимо отследить исполнение сценария "программа исполняется далее".
Смотрим в окно **RAM** (или в окно **SFR**).
В регистре **PortB** все биты установлены в **0** (в том числе и интересующий нас нулевой бит).
Следовательно, по логике команды **btfss**, следующей должна выполниться команда **goto PRD**, после чего рабочая точка программы "закольцуется" в ПП **PRD**, а это нас не устраивает (это не сценарий "программа исполняется далее").

Что делать?
Правильно, меняем **goto PRD** на все тот же "легендарный" **nop**.
В ПП **PRD** имеется две проверки.
Если отслеживается первая проверка, то со второй проверкой "возиться" нет смысла, так как она в точности такая же.
Следовательно, вторую проверку можно просто заблокировать (исключить из отслеживания), поставив перед обеими ее командами точки с запятой.
В тексте программы произошли изменения, значит производим ассемблирование (при этом, все точки остановки снимаются).
Устанавливаем программу на начало.
Назначаем точку остановки на команде **btfss PortB,0**, "запускаем автомат" и ждем окончания процесса.
После его окончания, рабочая точка программы установилась на команде **btfss PortB,0**.
Выполняем ее.
Выполняем следующую команду (**nop**).
Рабочая точка программы установилась на команде **goto START**, минуя команды второй проверки (они заблокированы).
Итак, мы "протащили" рабочую точку программы по выбранному сценарию ее работы (самому "протяженному". По количества команд).
Если выполнить команду **goto START**, то рабочая точка программы "улетит" на начало полного цикла программы.
Из рассмотренного выше, можно сделать следующие выводы:
Понятие "полный цикл программы" есть общее понятие. Его можно/нужно рассматривать как совокупность нескольких вариантов полных циклов программы (по количеству возможных сценариев ее исполнения).
По этой причине, не вполне корректный вопрос: "Каков полный цикл программы"?, нужно задавать иначе: "Каков полный цикл программы для конкретного сценария ее исполнения"?
Такая постановка вопроса корректна, ведь различные сценарии исполнения программы будут отличаться как по количеству команд, так и по своей функциональности.
Отслеживание выполнения программы (для выбранного ее сценария) можно производить не детально (а можно и детально, но это более трудоемко), а в "формализованном" виде.
Это означает то, что если имеются несколько одинаковых групп команд, то достаточно отследить выполнение одной из них, а другие проигнорировать
Конечно, все это достаточно условно и зависит от квалификации программиста.
И в самом деле, если программист "врезал" в текст программы подпрограмму (группу команд), с которой он много раз работал и которую знает "как облупленную", то какой смысл заниматься бестолковкой?
Например, в программе **cus**, группы команд формирования отрицательного и положительного полупериодов идентичны, и в принципе, можно отследить только выполнение одной из этих групп команд, а другую проигнорировать, но для этого нужно быть полностью уверенным в том, что в этой проигнорированной группе команд, нет ошибок.
То есть, опять же, такого рода упрощение работы "упирается" в квалификацию и опыт программиста.
Именно по этой причине, опытный программист может отследить выполнение различных сценариев работы программы "с реактивным свистом", а не имеющий такого опыта, может потратить на это дело уйму времени.
Все это - "дело наживное", и по мере "набора ума-разума", процесс отслеживания будет занимать все меньше и меньше времени.
Побольше "разборов полетов / тренировки", и все будет полном порядке.
Обращаю Ваше внимание на разницу между отслеживанием и отладкой программы.
Отслеживание выполнения программы первично.
Оно необходимо для проверки соответствия фактических алгоритмов ее исполнения задуманным, что автоматически означает проверку программы на предмет наличия ошибок функционального характера, которые, естественно, нужно устранить.
Отладка программы вторична.
Она производится после отслеживания ее выполнения (а какой смысл производить отладку, если есть вероятность того, что в тексте программы имеются функциональные ошибки?).
Проще говоря, слову "отслеживание" можно поставить в соответствие слово "грубо", а слову "отладка", слово "точно".

Вопрос: "Почему в тексте Самоучителя..., первой описывается отладка, а только затем, отслеживание"?
Ответ: на мой взгляд, такой порядок "въезда в проблему" наиболее рационален (в смысле доходчивости).
Итак, вернемся к процессу отслеживания работы программы **cus**.
Мы отследили только один сценарий ее работы. Переходим к другим.
Сценарии "порождаются" командами ветвления.
По количеству этих команд, можно составить представление об общем количестве сценариев исполнения программы.
По ходу отслеживания "основного" (такая классификация субъективна) сценария программы, в текст программы были "вставлены уловочные" **NOP**ы.
Совместим приятное с полезным: и оставшиеся сценарии отследим, и последовательно "избавимся" от "инородных" **NOP**ов (последовательно вернем текст программы в "исходное состояние").
Начнем с конца текста программы (так удобнее. Подумайте, почему?).
- Снимаем "блокировку" с команд второй проверки (убираем точки с запятыми).
- Убираем nop (после команды **btfss PortB,0**).
- Возвращаем на свое "штатное" место команду **goto PRD**.

Это означает то, что созданы условия для "закольцовки" рабочей точки программы в ПП **PRD**.
Проверяем это.
В текст программы внесены изменения, значит нужно произвести ассемблирование.
Производим.
Сбрасываем программу на начало.
Назначаем точкой остановки команду **btfss PortB,0**.
"Запускаем автомат" и ждем окончания его отработки.
Рабочая точка программы "встала" на назначенную точку остановки.
Выполняем команду **btfss PortB,0**.
В соответствии с логикой команды **btfss**, рабочая точка программы переместится на команду **goto PRD**.
Выполняем команду **goto PRD**.
Произойдет безусловный переход на первую команду ПП **PRD** ("врезку" **clrwdt**).
А теперь понажимайте клавишу **F7** "от души".
Произошла "закольцовка" ("уход в вечное кольцо") рабочей точки программы в ПП **PRD**, и пока **0**-й бит регистра **PortB** будет равен **0**, она так и будет там "крутиться", в чем и требовалось убедиться.
Проверяем следующий сценарий.
Убираем **nop** (после команды decfsz SecH,F) и возвращаем на свое "штатное" место команду **goto CYCLE**.
Ассемблируем, устанавливаем программу на начало, назначаем точку остановки на команде **decfsz SecH,F**.
"Запускаем автомат" и ждем окончания его отработки.
Рабочая точка программы "встала" на команду **decfsz SecH,F**.
Выполняем и ее, и следующую за ней команду **goto CYCLE**.
Рабочая точка программы переместилась на первую команду ПП **CYCLE** (**btfsc PortB,0**), и она будет "крутиться по кольцу" до тех пор, пока регистр **SecH** не очистится, в чем и требовалось убедиться.
Проверяем следующий сценарий.
Убираем **nop** (после команды **btfss PortB,6**) и возвращаем на свое "штатное" место команду **goto PRD**.
Ассемблируем, устанавливаем программу на начало, назначаем точку остановки на команде **btfss PortB,6**.
"Запускаем автомат" и ждем окончания его отработки.
Рабочая точка "встала" на команду **btfss PortB,6**.
Выполняем и ее, и следующую за ней команду **goto PRD**.
Рабочая точка программы переместилась на первую команду ПП **PRD** (**clrwdt**) и "закольцевалась" в ней, в чем и требовалось убедиться.
Итак, остался последний сценарий.
Заменяем команду **btfsc PortB,0** на команду **btfss PortB,0** (***еще одна "уловка" - берите на***

заметку).
Предлагаю Вам самостоятельно разобраться в сути произведенной замены (подсказка: комплиментарные операции).
Ассемблируем, устанавливаем программу на начало, назначаем точку остановки на команде **btfss PortB,0**.
Запускаем "автомат" и ждем окончания его отработки.
Рабочая точка "встала" на команду **btfss PortB,0**.
Выполняем и ее, и следующую за ней команду **goto START**.
Рабочая точка программы переместилась на первую команду ПП **START** (**clrf IntCon**) и далее, "закольцовывается" в ней, в чем и требовалось убедиться.
Заменяем "подставу" **btfss PortB,0** на "штатную" команду **btfsc PortB,0**.
Ассемблируем текст программы.
Итог: ***все сценарии работы программы cus отслежены, и текст программы "приведен в исходное состояние".***
Примечание: при отслеживании последних сценариев работы программы **cus**, точки остановки, конечно же, можно назначать и непосредственно на командах переходов.
А если говорить в общем, то для точки остановки, в пределах рабочей части программы, нет никаких ограничений.
В том смысле, что точкой остановки можно назначить любую команду.
Итак, я "протащил по программе" не только ее рабочую точку, но и Вас.
Почувствовали "почем фунт лиха"?
Каковы ощущения?
Предполагаю, что не типа "райское наслаждение", а несколько иные.
Спешу успокоить: "первый бой, он трудный самый".
По мере набора опыта, "дышать будет легче".
Это как с непривычки в парилку зайти: поначалу плоховато, затем начинает нравится, а потом "и за уши не оттащишь".
А если серьезно, то программирование в "чистом" ассемблере, это "епархия" упертых, въедливых и вовсе не пугливых людей.
О "манне небесной" тут и речи быть не может.
Прежде чем добраться до "конфетки", "попахать" придется как следует.
В первую очередь, именно по этой причине, программирование м/контроллеров, в глазах основной массы людей, является чем-то "заумным и недостижимым", хотя, в принципе, все "упирается" в разумно организованную и упорную работу, мало чем отличающуюся от любой другой работы, только со своей спецификой ("упор на мозги").

Пояснение

В этом разделе, применяются словосочетания "отрицательный полупериод" и "положительный полупериод".
При этом имеется ввиду модуляционный вход передатчика (см. разделительный конденсатор).
Если разделительного конденсатора нет, то эти словосочетания некорректны и нужно использовать словосочетания "интервал времени формирования нулевого уровня" и "интервал времени формирования единичного уровня".

11. Прерывания. Стек. Пример разработки программы с уходом в прерывания.

Описываемые, в этой книге, устройства и программы под них, не следует рассматривать как предложение создать такие устройства (хотя, это и вполне можно сделать, сымитировав внешние сигналы при помощи кнопок или переключателей), хотя бы по той простой причине, что не все "имеют под рукой" аппаратуру, к которой эти устройства должны подключаться.
Основная цель "разбора полетов", с привлечением этих устройств и программ, обучающая.
Та терминология, которая применяется, есть "продукт" моей фантазии ("рабочая точка программы", "закольцовка", "вечное кольцо", "уловка", "врезка", "шапка" и т.д.).
И далее будут придуманы и применены слова или короткие фразы, имеющие "привязки" к тем или иным понятиям.
Все это делается, как говорится, "не от хорошей жизни".
А что еще остается делать, если стандартная терминология либо "туманна", либо настолько неудобна и длинна, что если ей пользоваться, то к концу, можно забыть про начало.
Можете придумать свою терминологию. Это дело второе.
Главное → понимать суть происходящего, а терминология, это всего-лишь "обслуга" сути.
Разбираем "технологию" конструирования следующего устройства и программы под него.
Рассмотрим такую жизненную ситуацию (из практики моей работы): за большие деньги, закуплены дорогостоящие ретрансляторы Vertex-7000VXR.
Как это частенько бывает, когда техническими вопросами занимаются люди далекие от техники, после закупки выяснилось, что они однонаправленные.
То есть, "гонят информацию" только в одну сторону (с одной частоты на другую, а "обратного хода" нет), а необходим двунаправленный ретранслятор.
Если создавать такой ретранслятор из двух однонаправленных, то, кроме "вбухивания" в это дело большущих денег, возникает "куча" технических проблем.
Нужно было "спасать положение", а заодно и "прикрыть попу" начальству.
Следовательно, возникла задача: перевести однонаправленный ретранслятор Vertex-7000VXR в двунаправленный режим работы, с минимальными материальными затратами (себестоимость устройства - не более 250 руб.) и без потери качества ретрансляций.
Начинаем "борьбу с неопределенностями".
Сначала нужно "привязаться" к аппаратуре.
В ретрансляторе Vertex-7000VXR, не нужно организовывать переключение с приема на передачу, так как это происходит автоматически (в зависимости от наличия или отсутствия несущей).
Таким образом, задача сводится только к изменению "раскладки" частот: при ретрансляции в одну сторону, "раскладка" частот должна быть **ПРМ - X, ПРД - Y**, а при ретрансляции в другую сторону - наоборот, то есть**, ПРМ - Y, ПРД - X**.
Следовательно, речь идет о так называемом сканировании частот, то есть, о периодической смене раскладки частот, с остановкой их смены (остановкой сканирования), в интервале времени наличия несущей.
Это должно выглядеть так: при отсутствии несущей, сканирующее устройство должно поочередно переключать направления ретрансляции, то есть, указанные выше "раскладки" частот.
При этом, каждая из этих "раскладок" должна фиксироваться в течении некоторого оптимального интервала времени (слишком быстро или слишком медленно менять их нельзя).
При появлении несущей (естественно, в приемном тракте), в интервале времени отработки любой из этих "раскладок" частот, сканирование должно быть остановлено на время наличия несущей и возобновлено при ее пропадании.
Если несущая появляется в приемном тракте, настроенном на частоту **X**, то ретрансляция происходит в одном направлении, а если она появляется в приемном тракте, настроенном на частоту **Y**, то ретрансляция происходит в другом направлении.
Сам процесс ретрансляции нас "не волнует", так как этим "занимается" ретранслятор.
Ретранслятор Vertex-7000VXR, на задней стенке, имеет разъем, с выводов которого можно переключать раскладки частот (конкретные раскладки частот нужно заранее программировать), вывод сигнала управления шумоподавителем (несущая есть/нет = 0/1 соответственно), вывод +13,8v и корпус.
Выход, с которого снимается управляющий сигнал "несущая есть/нет" → с открытым коллектором.

Входы переключения частот рассчитаны на стандартные 5-вольтовые уровни (преобразователей уровней не нужно).
Таким образом, для подключения сканера к ретранслятору, имеются все необходимые цепи, компактно выведенные на разъем.
Теперь можно перейти к составлению блок-схемы программы.
Как было указано выше, при наличии несущей, сканирование должно быть остановлено.
Следовательно, речь идет о необходимости "ухода" рабочей точки программы (на время наличия несущей) в "вечное кольцо", с последующим выходом из него по внешнему воздействию, технология которого была описана ранее.
Для того чтобы не повторяться, я "закольцую" рабочую точку программы в так называемой подпрограмме прерывания.
"Учебно-тренировочная" необходимость в этом, диктуется тем, что "разборки" с прерываниями архинеобходимы (прерывания очень востребованы).
Сделаем так: сначала разберемся с прерываниями и стеком, затем "разложим на молекулы" программу с уходом в прерывания, и после этого, я покажу Вам "конструкцию" той же самой программы, но без ухода в прерывания.
Итак, чтобы двигаться дальше, нужно понять: что такое **прерывания**?
Для начала, отсылаю Вас во 2-й раздел (см. регистр **INTCON**).
К информации, имеющейся там, можно добавить следующее.
Если рассуждать поверхностно, то в прерывания можно вообще не уходить (вообще их не использовать), так как все то, что можно сделать в подпрограмме прерываний, можно сделать и в **"основном теле" программы** (это словосочетание – моя фантазия), организовав периодические проверки значений внешних (по отношению к ПИКу) управляющих сигналов.
Организовав уходы в прерывания, такого рода проверки делать не нужно. Если выбран внешний источник прерываний, то момент ухода в прерывание "привязан" к активному перепаду сигнала, который формирует этот источник.
По этой причине, отклик на внешнее воздействие будет гораздо "шустрее", нежели в случае работы без уходов в прерывания.
Определимся с терминологией.
Для удобства объяснения и восприятия, целесообразно разделить рабочую часть программы на две части.
***Условимся называть группу команд, которая отрабатывается после ухода в прерывание, как* подпрограмму прерывания**, ***а все остальное* → "основным телом" программы**.
"Технология" ухода в ПП прерывания следующая: например, мы решили "прерываться" по входу **RB0/INT** (в PIC16F84A, есть и другие источники прерываний → см. распечатку регистра **INTCON**).
Ввожу понятие **"зона" разрешения прерываний.**
Это "зона" представляет собой группу команд "основного тела" программы, ограниченная сверху, командой разрешения прерываний, а снизу, их запрета.
Если активный перепад внешнего, управляющего сигнала, присутствующего на выводе **RB0/INT**, сформируется в интервале времени отработки "зоны" разрешения прерываний, то рабочая точка программы "прыгнет" на начало ПП прерывания, после чего эта подпрограмма начнет отрабатываться.
В идеале, "зона" разрешения прерываний должна включать в себя всё "основное тело" программы, но специфика данного устройства такова, что возможен "ложный уход" в прерывание во время перестройки синтезатора частот, которая, после переключения раскладок частот, происходит в течение некоторого интервала времени (синтезатор частот перестраивается "не за ноль секунд"), чего допустить нельзя.
Это конечно не "высший сорт", но он сейчас и не нужен.
Сейчас нужно, на относительно простом примере, разобраться со смыслом и "технологией" этого "действа", а "высший сорт" будет потом.
ПП **START** должна начаться с команды запрета всех прерываний **clrf IntCon**.
Уходы в прерывания должны происходить позднее.
Далее, программа отрабатывается (исполняется) до начала "зоны" разрешения прерываний.
В начале этой "зоны", прерывания по входу **RB0/INT** разрешаются, после чего программа исполняется далее, если так можно выразиться, "в режиме ожидания прерываний".

В конце этой "зоны", прерывания запрещаются, и рабочая точка программы переходит на новый "виток" полного цикла "основного тела" программы.
В "зоне" разрешения прерываний, могут наступить 2 события:
- если прерывания нет, то программа исполняется в пределах "основного тела" программы,
- если прерывание есть (на выводе **RB0/INT** сформировался активный перепад), то происходит переход рабочей точки программы на начало исполнения ПП прерываний.

Такой переход происходит **по стеку**.
То есть, в стек записывается адрес следующей, после команды, во время исполнения которой произошло прерывание, команды.
Это нужно для того, чтобы после отработки ПП прерывания, произошел возврат на эту "следующую" команду.
Таким образом, рабочая точка программы, на некоторое время, как бы "отлучается" из "основного тела" программы.
На время этой "отлучки", отработка "основного тела" программы приостанавливается, а после возврата, снова возобновляется.
Например, представьте себе, что "зона" разрешения прерываний содержит 3 команды: **A, B** и **C** (по тексту программы, сверху вниз).
В том случае, если активный перепад управляющего сигнала сформировался во время исполнения команды **A**, то сразу же после исполнения этой команды, рабочая точка программы "улетает" в ПП прерывания.
При этом, адрес команды **B** автоматически запишется в стек и будет в нем находиться до тех пор, пока не исполнится последняя команда ПП прерываний, которой всегда должна быть команда **retfie**.
Она используется только для возврата из ПП прерывания.
По этой команде, осуществляется **возврат по стеку**.
Это означает то, что из него автоматически извлекается адрес команды **B** и возврат рабочей точки программы произойдет именно на эту команду, после чего, в "основном теле" программы, программа будет исполняться далее.
Если активный перепад управляющего сигнала сформировался во время исполнения команды **B**, то в стек запишется адрес команды **C**.
Ну и так далее.
Учтите, что программист не может напрямую управлять стеком.
Он может только косвенно воздействовать на него.
По этой причине, он является как бы "темной лошадкой".
Стек "подчиняется" только командам условных переходов и возвратов, а программиста "он просто ставит перед фактом события", за что его многие недолюбливают и стремятся обойтись без его привлечения (использование одних только команд безусловных переходов).
Однако, и в первую очередь это касается сложных программ, существует множество случаев, когда без задействования стека либо вообще не обойтись, либо оно (задействование) позволяет составить программу с гораздо меньшим количеством команд, чем в случаях, когда стек не задействуется.
Как не "увиливай" от этой "темной лошадки", но в тех случаях, когда речь идет о прерываниях, стек проигнорировать никак нельзя, так как он задействуется.
А раз это так, то где же команда **call** ?
При уходе в любой вид прерываний, команда* call *существует (а иначе, как работать со стеком?), ***но она выражена в неявном виде*** (в тексте программы ее нет), ***так как формируется не программно, а аппаратно.***
"Отмашкой" на формирование такой "неявной" команды является факт возникновения прерывания.
Более подробно, эта тема будет "провентилирована" позднее.
И еще одна очень важная "деталь": в состав ПП прерывания, в обязательном порядке, должна входить команда сброса флага задействованного источника прерываний (или флагов, если таких источников несколько).
В нашем случае, используется прерывание по входу **RB0/INT**.
В этом случае, команда сброса флага должна выглядеть так: **bcf IntCon,1**.
В большинстве случаев, команду (команды) сброса флага (флагов) прерывания помещают непосредственно перед командой **retfie**.

При отсутствии команды сброса флага прерывания, рабочая точка программы, после ухода в первое прерывания, начнет неконтролируемо "метаться" между ПП прерывания и "основным телом" программы (программа "уходит в глюк").
Проще говоря, если флаг прерывания "принудительно" (программно) не сброшен, то это "воспринимается" ПИКом как управляющий сигнал "ухода" в ПП прерывания.
Вывод: ***на момент возврата из ПП прерывания, флаг (флаги) прерывания должен быть программно сброшен (установлен в 0).***
Специалисты Микрочипа рекомендуют, в начале ПП прерывания, сохранить содержимое регистров **STATUS** и **W** в специально созданных, под это дело, регистрах общего назначения (стандартные названия: **Stat_Temp** и **W_Temp**, но они могут быть и другими), а в конце ПП прерывания, восстановить их содержимое.
Дело в том, что по ходу исполнения ПП прерывания, содержимое регистров **STATUS** и **W** может измениться, и после возврата из ПП прерывания, это содержимое может негативным образом повлиять на исполнение программы в "основном теле" программы.

В части касающейся заданного (прерывания по входу RB0/INT), для организации работы с прерываниями, необходимо:

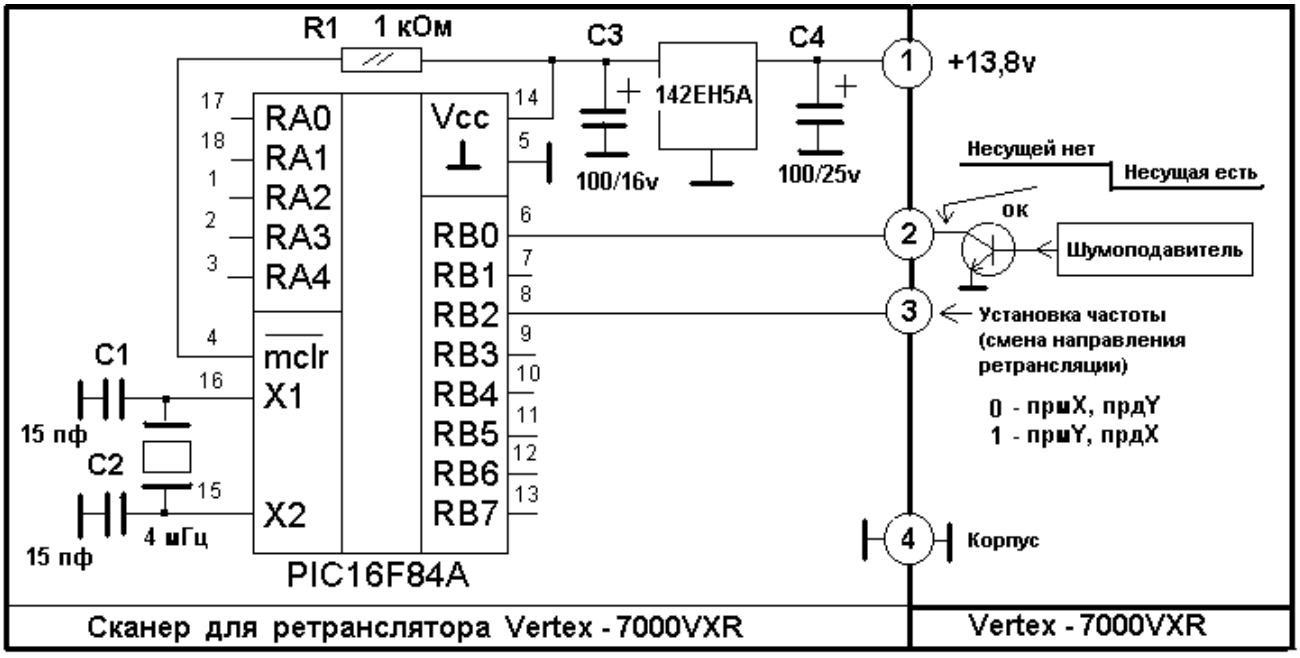
- в "шапке" программы, назначить вектор прерывания,
- назначить источник прерывания (INTCON),
- определить активный фронт прерывания (OPTION),
- создать "зону" разрешения прерываний,
- в начале ПП прерывания, сохранить, а в конце ПП прерывания, восстановить содержимое регистров STATUS и W,
- перед осуществлением возврата из ПП прерывания, сбросить флаг прерывания выбранного источника прерываний,
- в самом конце ПП прерывания, исполнить команду retfie.

Итак, уходим в прерывание.
Сразу возникает вопрос: "Что там будем делать"?
Исходя из того, что рабочая точка программы должна находиться в ПП прерывания все время, пока приемник фиксирует наличие несущей (управляющий сигнал "несущая есть"), вывод напрашивается сам по себе: в ПП прерывания, нужно "закольцевать" рабочую точку программы в "вечном кольце", то есть, сделать то, чем мы уже занимались раньше.
При этом, нужно организовать проверку типа "несущая есть/нет?" (опрос клавиатуры), с уходом рабочей точки программы в циклическую ПП задержки (при наличии несущей), основой которой является многоразрядный счетчик, и выходом из нее при "пропадании" несущей.
Ранее мы рассматривали работу вычитающего счетчика.
Теперь же, в обучающих целях, а заодно и для разнообразия, сделаю этот счетчик комбинированным, то есть, вычитающе - суммирующим (забегаю вперед: счетчик - двухразрядный).
В "основном теле" программы, сначала, нужно произвести подготовительные операции (ПП **START**).
Необходимо чем-то переключать направления ретрансляции, следовательно, нужно создать нечто типа "триггера", который управлял бы сменой "раскладок" частот.
Состояние этого "триггера" должно периодически опрашиваться, и в зависимости от этого, должен быть осуществлен переход на ту или иную раскладку частот.
Пусть опрос состояния "триггера" будет производиться сразу после ПП **START**, а изменение его состояния - в конце цикла "основного тела" программы (а можно и наоборот).
Между этим опросом и этим изменением, в простейшем случае (без учета специфики управляемого устройства), нужно сформировать "сплошную зону" разрешения прерываний, но этого делать нельзя по следующей причине.
"Раскладки" частот переключаются в синтезаторе частот ретранслятора Vertex-7000VXR, который, также как и любой синтезатор частот, является инерционным устройством.
То есть, переход с одной раскладки частот на другую, занимает некоторое время, и в это время возникают переходные помехи, наличие которых обусловлено переходными процессами в кольце фазовой автоподстройки частоты синтезатора частот.

Эти помехи могут "трансформироваться" в "паразитный" сигнал управления "несущая есть", что может привести к уходу в прерывания без фактического наличия несущей.
Для того чтобы избавиться от этой неприятности, нужно "переждать", пока не закончатся переходные процессы в кольце фазовой автоподстройки частоты синтезатора частот.
Таким образом, речь идет о том, что после выбора направления ретрансляции (переключения "раскладок" частот), необходимо, на некоторое время, "закольцевать" рабочую точку программы в ПП задержки.
То есть, сначала нужно сформировать некий защитный интервал времени, а только после этого формировать "зону" разрешения прерываний.
В целях обеспечения приемлемой инерционности обнаружения несущей, "зона" разрешения прерываний также должна иметь некоторую протяженность во времени.
Следовательно, в этой "зоне" должна располагаться ПП задержки с калиброванным временем задержки.
Если в течение этого времени на выводе **RB0/INT** возникнет активный перепад (несущая есть), то процесс сканирования прекратится, так как рабочая точка программы "улетит" в ПП прерывания и будет "крутиться" там в "вечном кольце" до тех пор, пока не "пропадет" несущая.
Если это случится, то произойдет возврат из прерывания, и сканирование возобновится.
Теперь можно составлять блок - схему программы:

И принципиальную схему устройства тоже можно составить.
Определяемся с выводами портов.

Порт А не задействуем, порт В задействуем.
Входом внешнего прерывания **INT** является вывод **RB0**, следовательно, вывод **RB0/INT** необходимо настроить на работу "на вход".
Примечание: никакой другой вывод порта (любого) нельзя назначить входом внешнего прерывания **INT**, так как этот вид прерываний "жестко привязан" именно к выводу **RB0**.
Так как управляющий сигнал "несущая есть/нет" поступает на вывод **RB0/INT** с выхода каскада с открытым коллектором, то необходимо включить подтягивающие резисторы порта В.
В этом случае, подтягивающий резистор вывода **RB0/INT** будет являться коллекторной нагрузкой каскада с открытым коллектором.
Экспериментально проверяю состояние каскада с открытым коллектором: при отсутствии несущей транзистор закрыт, а при ее наличии, открыт, что с учетом подтягивающего резистора вывода **RB0/INT**, соответствует единичному уровню, если несущей нет, и нулевому уровню, если она есть.
Берем на заметку. При составлении программы, это потребуется.
Выходом управления переключением направлений ретрансляций (раскладки частот) можно назначить любой из оставшихся выводов порта В.
Пусть им будет, например, вывод **RB2**.
Этот вывод нужно настроить на работу "на выход".
Остальная часть принципиальной схемы практически такая же, как у устройства формирования тонального вызова с частотой 1450 Гц.



Пояснение: в ретрансляторе Vertex-7000VXR, для дистанционного переключения пар частот (одна частота - приемная, другая - передающая), на разъем выведено 4 цепи (переключение пар частот производится в инвертированном, весовом, параллельном коде).
Таким образом, можно осуществить переключения между 16-ю парами частот, но нам необходимо переключаться только между двумя парами частот, следовательно, под это дело, можно задействовать только один младший разряд (он и подключен к выводу **RB2**), а остальные 3 разряда запараллелить и "подать" на них единицу, то есть, подключить их к 4-му выводу **PIC16F84A.**
В этом случае, переключения будут производиться между 1-м и 2-м каналами (1-й и 2-й парами частот).
1-му каналу соответствует единица на выводе **RB2**, а 2-му каналу, ноль на выводе **RB2**.
В принципиальной схеме, с целью упрощения ее восприятия, я не показал описанных выше соединений (но они должны быть).
Просто имейте это ввиду.
Принципиальную схему устройства вообще можно предельно упростить, если запитаться от цепи +5v ретранслятора.

В этом случае, стабилизатор на 142ЕН5 можно убрать, но придется "лезть" в ретранслятор и делать отвод от +5v.
А теперь можно прикинуть стоимость этого устройства (25-штырьковый разъем - компьютерный).
В 250 руб. уложился (по ценам на комплектующие, которые имели место быть на момент "ваяния этой железяки").

Файл текста программы под это устройство называется Retr_1.asm (находится в папке ***"Тексты программ"***).

Программа выглядит так:

```
;*******************************************************************************
; Retr_1.asm                    ВАРИАНТ С ПРЕРЫВАНИЯМИ
;                    Сканер для ретранслятора VERTEX-7000VXR.
; Автор: Корабельников Евгений Александрович  г.Липецк  декабрь 2004г.
; E-mail: karabea@lipetsk.ru                 http://ikarab.narod.ru
;*******************************************************************************
; Позволяет перевести VERTEX-7000VXR и другие подобные ретрансляторы из режима
; односторонней ретрансляции в режим двухсторонней ретрансляции без потери
; качества работы.
;*******************************************************************************
; Используется микроконтроллер PIC16F84A. Частота кварца 4000кГц.
;*******************************************************************************
            LIST        p=16F84a     ; Используется PIC16F84A.
            __CONFIG    03FF5H       ; WDT включен, бит защиты не установлен.
;===============================================================================
; Определение положения регистров специального назначения.
;===============================================================================
OptionR     equ         01h          ; Option - банк1
Status      equ         03h          ; Регистр Status
PortB       equ         06h          ; Порт B
TrisB       equ         06h          ; Tris B - Банк1
IntCon      equ         0Bh          ; Регистр IntCon
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
Trigg       equ         0Ch          ; Переключатель направления ретрансляции.
W_Temp      equ         0Eh          ; Регистр сохранения содержимого W в
                                     ; прерываниях.
Stat_Temp   equ         0Fh          ; Регистр сохранения содержимого STATUS в
                                     ; прерываниях.
SecH        equ         1Eh          ; Старший байт счетчика времени сканирования.
SecL        equ         1Fh          ; Младший байт счетчика времени сканирования.
SecH_1      equ         1Ch          ; Старший байт счетчика защитного интервала
                                     ; времени.
SecL_1      equ         1Dh          ; Младший байт счетчика защитного интервала
                                     ; времени.
SecH_2      equ         1Ah          ; Старший байт счетчика задержки рабочей точки
                                     ; программы в прерывании.
SecL_2      equ         1Bh          ; Младший байт счетчика задержки рабочей точки
                                     ; программы в прерывании.
;===============================================================================
; Определение  места  размещения результатов операций.
;===============================================================================
W           equ         0            ; Результат направить в аккумулятор.
F           equ         1            ; Результат направить в регистр.
;===============================================================================
; Определение положения флагов и бита выбора банка в регистре STATUS.
;===============================================================================
RP0         equ         5            ; Бит выбора банка.
;===============================================================================
; Точка входа в программу.
;===============================================================================
```

```
            org         0           ; Начать выполнение программы с нулевого
            goto        START       ; адреса PC.
;=============================================================================
; Точка входа в прерывание.
;=============================================================================
            org         4           ; Назначение вектора прерывания (назначается,
                                    ; если в программе используются прерывания).
;=============================================================================
; Объем программы: 63 слова в памяти программ.
;*****************************************************************************

;*****************************************************************************
; НАЧАЛО ПРЕРЫВАНИЯ.
;=============================================================================
; Сохранение  содержимого регистров STATUS и W в ОЗУ.
;-----------------------------------------------------------------------------
INT         movwf       W_Temp      ; Скопировать содержимое регистра W
                                    ; в регистр W_Temp.
            movf        Status,W    ; Скопировать содержимое регистра Status
                                    ; в регистр W.
            movwf       Stat_Temp   ; Скопировать содержимое регистра W
                                    ; в регистр Stat_Temp.
;-----------------------------------------------------------------------------
; Проверка наличия несущей (опрос клавиатуры).
;-----------------------------------------------------------------------------
CYCLE       btfsc       PortB,0     ; Проверка состояния нулевого бита
                                    ; регистра PortB.
            goto        EndInt      ; Если он =1 (несущей нет), то уход
                                    ; в ПП выхода из прерывания.
                                    ; Если он =0 (несущая есть), то программа
                                    ; исполняется далее.
;-----------------------------------------------------------------------------
; Формирование времени одного цикла задержки рабочей точки программы в прерывании
;-----------------------------------------------------------------------------
            movlw       .250        ; Закладка константы .250 в регистр W.
            movwf       SecH_2      ; Копирование .250 из регистра W
                                    ; в регистр SecH_2.
            movlw       .120        ; Закладка константы .120 в регистр W.
            movwf       SecL_2      ; Копирование .120 из регистра W
                                    ; в регистр SecL_2.
PAUSE_2     clrwdt                  ; Сброс WDT.
            decfsz      SecL_2,F    ; Декремент содержимого младшего разряда
                                    ; счетчика SecL_2.
            goto        PAUSE_2     ; Если результат декремента не =0, то переход
                                    ; в ПП PAUSE_2.
            incfsz      SecH_2,F    ; Если результат декремента =0, то инкремент
                                    ; старшего разряда счетчика SecH_2.
            goto        PAUSE_2     ; Если результат инкремента не=0, то переход
                                    ; в ПП PAUSE_2.
            goto        CYCLE       ; Если результат инкремента =0, то переход на
                                    ; следующий цикл задержки рабочей точки
                                    ; программы в прерывании.
;=============================================================================
; Восстановление содержимого регистров STATUS и W с последующим выходом
; из прерывания.
;-----------------------------------------------------------------------------
EndInt      bcf         IntCon,1    ; Сброс флага прерывания по INT.
            movf        Stat_Temp,W ; Скопировать содержимое регистра Stat_Temp
                                    ; в регистр W.
            movwf       Status      ; Скопировать содержимое регистра W
                                    ; в регистр Status.
            swapf       W_Temp,F    ; Поменять местами старший и младший полубайты
                                    ; регистра W_Temp с сохранением результата
                                    ; операции в нем же.
            swapf       W_Temp,W    ; Поменять местами старший и младший полубайты
                                    ; регистра W_Temp с сохранением результата
```

```
                                    ; операции в регистре W.
            retfie                  ; Возврат из прерывания по стеку.
;*****************************************************************************
; КОНЕЦ ПРЕРЫВАНИЯ
;*****************************************************************************

;*****************************************************************************
; СКАНИРОВАНИЕ
;*****************************************************************************
; Подготовительные операции.
;-----------------------------------------------------------------------------
START       clrf        IntCon      ; Запретить все прерывания.
            clrwdt                  ; Сбросить сторожевой таймер WDT.
            bsf         Status,RP0  ; Установить банк 1.
            movlw       .1          ; RB0 работает на вход,
            movwf       TrisB       ; остальные - на выход.
            movlw       .0          ; Включить подтягивающие резисторы порта B.
            movwf       OptionR     ; На входе INT прерывания - по заднему фронту.
            bcf         Status,RP0  ; Установить банк 0.
;-----------------------------------------------------------------------------
; Выбор направления ретрансляции.
;-----------------------------------------------------------------------------
TRIGGER     btfsc       Trigg,0     ; Проверка значения нулевого бита
                                    ; регистра Trigg.
            goto        Metka148    ; Если это значение =1, то переход
                                    ; на метку Metka148.
                                    ; Если это значение =0, то программа
                                    ; исполняется далее.
            movlw       .251        ; Закладка константы .251 (1111 1011)
                                    ; в регистр W.
            movwf       PortB       ; Копирование .251 из регистра W
                                    ; в регистр PortB (выбор прямого направления
                                    ; ретрансляции: ПРМ-X, ПРД-Y).
            goto        Metka_1     ; Безусловный переход на метку Metka_1.
Metka148    movlw       .255        ; Закладка константы .255 (1111 1111)
                                    ; в регистр W.
            movwf       PortB       ; Копирование .255 из регистра W
                                    ; в регистр PortB (выбор обратного направления
                                    ; ретрансляции: ПРМ-Y, ПРД-X).
;-----------------------------------------------------------------------------
; Формирование защитного интервала времени (ожидание окончания переходных
; процессов, возникающих при смене направлений ретрансляции) равного,
; примерно, 60 мс.
;-----------------------------------------------------------------------------
Metka_1     movlw       .197        ; Закладка в регистр W константы .197
            movwf       SecH_1      ; Копирование константы .197 из регистра W
                                    ; в регистр SecH_1.
            movlw       .121        ; Закладка в регистр W константы .121
            movwf       SecL_1      ; Копирование константы .121 из регистра W
                                    ; в регистр SecL_1.
PAUSE_D     clrwdt                  ; Сброс WDT.
            decfsz      SecL_1,F    ; Декремент содержимого младшего разряда
                                    ; счетчика SecL_1.
            goto        PAUSE_D     ; Если результат декремента не =0,
                                    ; то переход в ПП PAUSE_D.
            incfsz      SecH_1,F    ; Если результат декремента =0, то инкремент
                                    ; старшего разряда счетчика SecH_1.
            goto        PAUSE_D     ; Если результат инкремента не=0,
                                    ; то переход в ПП PAUSE_D.
                                    ; Если результат инкремента =0, то программа
                                    ; исполняется далее.
;=============================================================================
; Формирование интервала времени сканирования.
;=============================================================================
            movlw       .245        ; Закладка константы .245 в регистр W.
            movwf       SecH        ; Копирование .245 из регистра W
```

```
                                ; в регистр SecH.
            movlw       .255    ; Закладка константы .255 в регистр W.
            movwf       SecL    ; Копирование .255 из регистра W
                                ; в регистр SecL.
;-------------------------------------------------------------------------------
; Разрешение прерываний INT.
;-------------------------------------------------------------------------------
            movlw       .144    ; Закладка константы .144 в регистр W.
            movwf       IntCon  ; Копирование .144 (1001 0000) из регистра W
                                ; в регистр IntCon
                                ; (разрешение прерываний по входу INT).
;-------------------------------------------------------------------------------
PAUSE       clrwdt              ; Сброс WDT.
            decfsz      SecL,F  ; Декремент содержимого младшего разряда
                                ; счетчика SecL.
            goto        PAUSE   ; Если результат декремента не=0,
                                ; то переход в ПП PAUSE.
            incfsz      SecH,F  ; Если результат декремента =0, то инкремент
                                ; старшего разряда счетчика SecH.
            goto        PAUSE   ; Если результат инкремента не=0,
                                ; то переход в ПП PAUSE.
                                ; Если результат инкремента =0,
                                ; то программа исполняется далее.
;-------------------------------------------------------------------------------
; Изменение направления ретрансляции.
;-------------------------------------------------------------------------------
            clrf        IntCon  ; Запрет всех прерываний.
            incf        Trigg,F ; Увеличение на 1 (инкремент) содержимого
                                ; переключателя направления ретрансляции.
            goto        START   ; Переход на новый цикл сканирования.
;*******************************************************************************
            end                 ; Конец программы.
```

Создайте проект с названием **Retr_1**, загрузите в него файл **Retr_1.asm**, проассемблируйте текст программы и убедитесь в отсутствии ошибок.
Обращаю Ваше внимание на то, что при использовании прерываний, директива назначения вектора прерываний **org 4** (эту директиву просто нужно принять как данность) должна находиться, в "шапке" программы, сразу же за командой **goto START**.
То есть, после возникновения факта прерывания, рабочая точка программы автоматически "улетит" в ячейку **PC** с адресом **04h** (это определено разработчиками ПИКов).
Откройте окно **ROM** и убедитесь в этом.
Если после директивы **org 4**, нет команды перехода в ПП прерывания, то рабочая часть программы должна начинаться с ПП прерывания, а "основное тело" программы должно быть расположено сразу же после команды **retfie**.
Если ПП прерывания "врезана" в "основное тело" программы (располагается, например, в середине текста программы. Условно), то такой переход должен быть.
То есть, в ячейке PC с адресом 04h, должна находиться
- **либо первая команда ПП прерывания,**
- **либо команда перехода в ПП прерывания.**

В данном случае, речь пойдет о первом варианте.
Вопросы может также вызвать и ПП **INT**.
Вопрос: "В тексте программы, команд переходов, в эту ПП (или на эту метку), нет (они и не нужны, так как переход происходит по вектору прерываний, которому "по барабану" все названия). Для чего нужно это название"?
Ответ: для удобства отслеживания и отладки.
Пояснение.
В ходе работ по отслеживанию и отладке программы, с переходом в "основное тело" программы, проблем не возникает: сбрасываем программу на начало, нажимаем **F7** и попадаем на первую команду ПП **START**.
А как попасть на первую команду ПП прерываний?
Без "уловок" здесь не обойтись.

Самая простая из них: для обеспечения перехода на первую команду ПП прерывания (то есть, в ПП прерывания), на этой команде выставляется метка (в нашем случае **INT**).
А можно сказать и то, что подпрограмме присваивается название.
Можно и так, и эдак. На смысл не влияет.
В "шапку" программы, сразу же после команды **goto START**, "врезается" команда **goto INT** (в тексте программы ее нет, но при отслеживании и отладке программы ее нужно вставить).
А теперь все очень просто: если ставим точку с запятой (блокируем команду) перед командой **goto INT** (перед командой **goto START** точки с запятой нет), то попадаем на первую команду ПП **START**, а если снимаем точку с запятой с команды **goto INT** и ставим точку с запятой перед командой **goto START**, то попадаем на начало ПП прерывания.
Вот для этого и нужна метка с названием **INT** (или с другим названием).
В текст программы, я ее ввел для того чтобы обозначить начало ПП прерывания.
Команду **goto INT** можете прямо сейчас "врезать" в "шапку" программы, только заблокируйте ее точкой с запятой.
В дальнейшем, при отслеживании и отладке, она потребуется.
Перед "прошивкой", то есть, после того, как программа полностью отработана, эту метку (а также и команду **goto INT**), из текста программы можно удалить.
Это никак не повлияет на работу устройства.

Разбираем текст программы

Эту "разборку" я буду проводить не так детально, как ранее.
С учетом того, что предыдущая информация Вами усвоена.
Принцип "конструкции шапки" программы – стандартный.
Повторяться не буду. Остановлюсь только на особенностях.
Три пары регистров с названием **Sec...** - элементы трех двухразрядных счетчиков.
Регистры со стандартными (рекомендовано разработчиками) названиями **W_Temp** и **Stat_Temp**, это регистры оперативной памяти, использующиеся для сохранения содержимого регистров **W** и **Status**, в интервале времени исполнения ПП прерывания.
На регистре с названием **Trigg** "собран" триггер со счетным входом.
Так как используются команды, результат исполнения которых необходимо сохранить в регистре **W**, то в ***"Определении места размещения результатов операций"***, дополнительно "прописан" регистр **W**.
В соответствии со сказанным выше, в конце "шапки" программы, располагается директива **org 4**.
Выполнение программы начинается с запрета всех прерываний (**clrf IntCon**).
Команду сброса **WDT** (в программе он задействован. См. биты конфигурации) можно вставить в любое место подготовительных операций.
Причина такой "демократии" заключается в том, что интервал времени срабатывания **WDT** значителен, и он с запасом "перекрывает" интервал времени исполнения команд подготовительных операций (и не только их).
Если последующие команды **clrwdt** обеспечивают перезапуск RC-одновибратора сторожевого таймера до окончания формирования его импульса, то команда **clrwdt**, расположенная в начале исполнения программы, в принципе, не нужна, но "пусть живет" ("кашу маслом не испортишь").
Далее, также как и в программе **cus**, "настраиваются" направления работы выводов порта B: **RB0/INT** - "на вход", остальные (в том числе и интересующий нас вывод **RB2**) - "на выход" (**00000001**).
Подтягивающие резисторы порта B включены.
Собственно говоря, нужно подключить только один подтягивающий резистор.
К выводу **RB0/INT**.
Но так как они могут быть включены или отключены только "оптом",значит подключаем их все
В этом же регистре **OptionR**, **6**-й бит (выбор активного фронта прерывания **INT**) устанавливается в ноль.
То есть, в соответствии с задуманным (см. задание на разработку), прерывание будет осуществляться по перепаду от **1** к **0**.
Все остальные биты регистра **OptionR**, кроме **6**-го и **7**-го, нас не интересуют.
В них можно выставить любой уровень.
Для удобства, "привяжемся" к нулевым уровням **6**-го и **7**-го битов.

Поэтому, во всех битах регистра **OptionR**, выставляем нули, что соответствует числовому значению байта = **0**.
Смотрим в блок-схему программы.
Теперь нужно опросить состояние переключателя направлений ретрансляции типа "триггер со счетным входом".
Создаем этот триггер (или переключатель. Кому как нравится).
Процесс конструирования подобного рода устройств (всего-навсего 2 состояния: **0** или **1**) выглядит так.
Под это дело, назначается ("прописывается") регистр общего назначения.
В нашем случае, это регистр **Trigg** (см. "шапку" программы). Можно назвать и по-другому.
Все регистры общего назначения являются 8-битными.
Поэтому, диапазон чисел, который они способны отобразить: от **.0** до **.255**.
В данном случае, такое большое количество состояний нам ни к чему, ведь нужно только два из них.
Какие именно?
Любой из 8-ми двоичных разрядов может находиться в одном из двух состояний (**0** или **1**), но чаще всего эти состояния будут меняться (в результате инкремента или декремента) в младшем, двоичном разряде.
То есть, в этом случае, они будут меняться после каждого инкремента или декремента, а не "через раз" и более.
Последовательно инкрементируя или декрементируя содержимое регистра **Trigg**, значение бита его **младшего разряда** будет меняться с **0** на **1** или наоборот.
Вот Вам и триггер со счетным входом.
При этом, значения битов оставшихся 7-ми разрядов нас не интересуют. Вообще.
Таким образом, опрос состояния регистра **Trigg** можно производить "по принципу четности/нечетности": если в нем четное число (бит **№0**=**0**), то программа исполняется по одному сценарию, а если нечетное (бит **№0**=**1**), то программа исполняется по другому сценарию.
Применительно к нашему случаю, это означает то, что при равенстве нулевого бита регистра **Trigg** нулю, ретрансляция будет происходить в одну сторону, а при его равенстве единице, в другую сторону.
Разбираем ПП **TRIGGER**.
Первая ее команда проверяет состояние нулевого бита регистра **Trigg** (**btfsc Trigg,0**).
btfsc - команда ветвления, следовательно, имеются 2 сценария дальнейшей работы программы.

1. В случае исполнения одного сценария, в регистр **PortB** записывается константа **.251** (**11111011**), то есть, на выводе **RB2** устанавливается **0** (одно направление ретрансляции).
2. В случае исполнения другого сценария, в регистр **PortB** записывается константа **.255** (**11111111**), то есть, на выводе **RB2** устанавливается **1** (другое направление ретрансляции)

Это связано со спецификой управления конкретным типом ретранслятора.
Так как на "линейном участке" программы (а мы сейчас как раз с таким и работаем) рабочая точка программы движется только сверху вниз (по тексту программы) и последовательной записи обеих констант в регистр **PortB** допустить нельзя (должна записаться либо одна, либо другая), то для того чтобы это обеспечить, необходимо команду **movlw .255** пометить меткой (**Metka148**), и в одном из двух случаев, осуществить безусловный переход на эту метку.
При такой "конструкции" проверки, на команду **movlw .255**, никаким иным образом, кроме как с использованием команды перехода, перейти нельзя.
Кроме того, необходимо, чтобы после исполнения любого из этих сценариев, рабочая точка программы попала на одну и ту же команду продолжения программы: в нашем случае это команда **movlw .197**.
Это можно условно назвать "разветвлением на сценарии, с последующим их слиянием".
Обязательно обратите на это внимание, так как в дальнейшем, в ходе "въезда в выравнивание сценариев", это пригодится.
Если после исполнения команд записи константы **.255**, в регистр **PortB**, рабочая точка программы, "естественным образом попадает" на команду **movlw .197**, то после исполнения команд записи константы **.251**, в регистр **PortB**, она может "попасть" на команду **movlw .197** только после того, как произойдет запись константы **.255**, в регистр **PortB**.
Этого допустить нельзя.

Какой смысл записывать **.251**, если в конечном итоге, все равно запишется **.255** ?
Значит, если в регистр **PortB,** записывается константа **.251**, то последующие команды записи, в него, константы **.255,** нужно обойти ("перепрыгнуть" через них).
То есть, нужно пометить команду **movlw .197** еще одной меткой (**Metka_1**), и после исполнения команд записи константы **.251**, в регистр **PortB**, нужно на нее перейти (**goto Metka_1**).
Посмотрите, как это сделано в программе **Retr_1**.
После того, как направление ретрансляции выбрано (произошло переключение "раскладок" частот), необходимо сформировать защитный интервал времени продолжительностью примерно **50 … 150 мс.** (что это такое → см.выше).
Для Vertex-7000VXR, хватит и **60 мс.**
На эту цифру я и буду ориентироваться при отладке программы.
Таким образом, речь идет о банальной (без "выкрутасов") ПП задержки.
Следовательно, предварительно (то есть, до начала ее исполнения), нужно записать, в задействованные под нее регистры общего назначения (**SecL_1** и **SecH_1**) соответствующие константы.
Калиброванную задержку величиной порядка 60 мс. можно сформировать и одним разрядом счетчика, но с точки зрения универсальности устройства (оно может быть подключено и к аппаратуре с бОльшей инерционностью синтезатора частот), лучше сделать его двухразрядным.
Для разнообразия, в обучающих целях, младший разряд счетчика я сделаю вычитающим, а старший разряд - суммирующим.
Собственно говоря, добавить что-то к тому, что было сказано ранее (при "разборках" с двухразрядным, вычитающим счетчиком. Программа **cus**), нечего, за исключением напоминания о том, что в разряде суммирующего счетчика, направление "приближения" к нулю противоположно направлению "приближения" к нулю, в разряде вычитающего счетчика.
Если, например, нужно увеличить время задержки, то числовое значение константы вычитающего разряда счетчика необходимо увеличить, а числовое значение константы суммирующего разряда счетчика необходимо уменьшить.
Что касается конкретных, числовых значений констант, то они подобраны из расчета формирования защитного интервала времени величиной, примерно, **60 мс.**
Попробуйте, для тренировки, "грубо" рассчитать числовые значения констант, для защитного интервала времени = 60 мс. (с учетом "конструкции" счетчика), и сравните результат этого расчета с тем, что "имеется в наличии".
В итоге, Вы должны получить числовые значения констант, близкие к тем, которые указаны в тексте программы.
Теперь, в соответствии с блок-схемой программы, необходимо сформировать интервал времени сканирования.
В принципе, это делается точно так же, как и в случае формирования защитного интервала времени.
С целью недопущения "черепно-мозгового бардака", в счетчике интервала времени сканирования, я задействовал отдельную пару регистров общего назначения, с названиями **SecH** и **SecL**.
Таким образом, в "основном теле" программы, с которым мы сейчас работаем, задействованы две пары регистров общего назначения:
- при формировании защитного интервала времени: **SecH_1** и **SecL_1**,
- при формировании интервала времени сканирования: **SecH** и **SecL**.

А теперь посмотрим на содержимое регистров **SecH_1** и **SecL_1**, на момент начала формирования интервала времени сканирования.
Оба они очищены (в них записаны нули), и далее, по ходу исполнения программы (в том числе и в ПП прерывания), они не используются.
Вывод: их можно использовать повторно. При формировании интервала времени сканирования.
То есть, можно обойтись без регистров **SecH** и **SecL**, заменив их, соответственно, на **SecH_1** и **SecL_1**.
Верно также и обратное утверждение: можно оставить только **SecH** и **SecL**, а **SecH_1** и **SecL_1** убрать ("вычеркнуть" из "шапки" программы).
В данном случае, в пределах одного полного цикла программы, работа этой пары регистров, будет выглядеть так: после записи, в них, констант защитного интервала времени (в начале

формирования этого интервала) и очистки этих регистров (в конце формирования этого интервала), в них снова будут записаны константы, но только не защитного интервала времени, а интервала времени сканирования, после чего, в конце формирования интервала времени сканирования, эти регистры снова очистятся.
И это должно быть понятным, так как программа исполняется последовательно (сразу две задержки формироваться не могут).
С учетом сказанного, конечно же, в текст программы (если эту программу предполагается использовать не как учебную) можно внести изменения, но с точки зрения обеспечения минимальной путаницы, при "въезде" в программу, я посчитал целесообразным "расписать" именно этот (4 регистра вместо 2-х) не очень-то и рациональный вариант ее построения.
Практический вывод, из этих рассуждений, следующий: ***если в программе задействованы однотипные счетчики, то во многих случаях, в них можно использовать одну и ту же группу регистров общего назначения.***
Забегая вперед, расскажу про случай, когда одну и ту же группу регистров общего назначения нельзя использовать в двух счетчиках.
В ПП прерывания имеется, аналогичный описанным выше, двухразрядный счетчик на регистрах **SecH_2** и **SecL_2**.
Пары регистров **SecH_2**, **SecL_2** и **SecH_1, Sec_L1** можно объединить в одну пару, так как на момент начала работы одного счетчика, содержимое другого счетчика очищено (не значимо).
А вот регистры **SecH_2, Sec_L2** и **SecH, SecL** объединить нельзя, так как содержимое счетчика **SecH, SecL** значимо.
Если его уничтожить "записью "по верху", то нормальная работа устройства будет нарушена.
Детали этого утверждения будут понятны после "въезда" в работу ПП прерывания.
На первых порах, рекомендую Вам, под каждый счетчик, назначать отдельную группу регистров общего назначения (так, как это сделано в программе **Retr_1.asm**), а указанными выше объединениями заняться позднее. После приобретения опыта и навыков.
Теперь вернемся к счетчику интервала времени сканирования.
Его конструкция и принцип работы должны быть сейчас Вам понятны.
В соответствии со сформулированным ранее алгоритмом работы устройства, во время формирования интервала времени сканирования, рабочая точка программы должна (при наличии активного перепада на выводе **RB0**) уйти в ПП прерываний.
То есть, "зона" разрешения прерываний должна включать в себя именно этот интервал времени.
В начале этой "зоны", нужно разрешить внешние прерывания **INT**.
Разрешить их до исполнения команд (или между ними) записи констант, в регистры счетчика (**SecH, Sec_L**), можно, но в данном случае, я "привязался" непосредственно к началу счета.
Поэтому, команды записи константы, в регистр **IntCon**, расположены именно в том месте, где Вы их видите (см. текст программы **Retr_1.asm**).
В регистре **IntCon**, биты **№4** и **7** установлены в **1** (в остальных, **0**).
Это и есть разрешение прерываний **INT** (все остальные виды прерываний, которыми управляет регистр **IntCon**, запрещены нулями).
Общая особенность: ***при разрешении любого вида прерываний, бит глобального разрешения прерываний должен быть установлен в 1.***
Если прерывания **INT** разрешены (бит **№4** = **1**), а глобального разрешения прерываний нет (бит **№7** = **0**), то никаких прерываний Вы не дождетесь (в том числе и **INT**).
В конце "зоны" разрешения прерываний, прерывания нужно запретить.
Проще всего это сделать, запретив их все (**clrf IntCon**).
Запретить все прерывания также можно, установив в **0** один только **7**-й бит регистра **IntCon** (**bcf IntCon,7**), но зачем, на первых порах, "напрягаться"?
Проще сбросить весь байт в **0**, и дело с концом. Что и сделано.
А вообще, это личный выбор каждого.
Смотрим в блок - схему программы.
Теперь нужно обеспечить предварительную подготовку смены раскладки частот для следующего "витка" полного цикла программы.
При этом, содержимое регистра **Trigg** можно или декрементировать, или инкрементировать.
Так как речь идет о **проверке на четность** (используется только бит **№0**. Числовое значение байта может быть любым), то это "что в лоб, что по лбу".
Выбираю инкремент.
Самое главное, это ответ на вопрос: "Какое это число: четное (бит **№0** = **0**) или нечетное

(бит **№0** = **1**)"?
От ответа на этот вопрос зависит, в какой именно сценарий работы программы "улетит" ее рабочая точка.
От "витка" к "витку" полного цикла программы, будут происходить последовательные смены четных чисел на нечетные (или наоборот), и в соответствии с этим, будут меняться "раскладки" частот.
После того, как произведен инкремент (см. конец текста программы) содержимого регистра **Trigg** (**incf Trigg,F**), можно переходить на "новый", полный цикл "основного тела" программы (**goto START**).
Обратите внимание на то, что применена команда **incf**, которая не является командой ветвления.
В данном случае, "ветвиться" не нужно. Поэтому она и применена.
Итак, мы "прошли по основному телу" программы, исходя из предположения, что активных перепадов "ухода" в прерывание (по входу **RB0/INT**) не поступало.
Теперь предположим, что на вывод **RB0/INT**, от внешнего устройства, поступил активный перепад (от **1** к **0**).
Если это событие произошло во время отработки "зоны" разрешения прерываний, то произойдет условный переход ("виртуальная" команда **call**. Формируется аппаратно) на первую команду ПП прерывания (В PC, адрес **04h**).
При этом, в вершину стека будет скопировано числовое значение адреса команды, следующей за командой, во время исполнения которой имел место быть активный перепад.
"Закладка" этого адреса происходит не в момент фактического возникновения активного перепада, а после "полноценной" отработки той команды, в интервале времени исполнения которой этот активный перепад имел место быть.
После того как рабочая точка программы "улетит" на начало ПП прерывания, эта ПП будет исполнена. Вплоть до команды **retfie** (возврат из прерывания) включительно.
При этом, происходит возврат по ранее "заложенному", в стек, адресу, после чего этот адрес, по причине дальнейшей ненужности, "убивается" путем его безжалостного удаления из вершины стека.
Ироничное замечание: упомянутый выше, достаточно банальный, условный переход (пусть даже и команда **call** "виртуальная") называется "переходом по вектору прерывания".
Любят у нас мудрить без меры …
Вот так и возникает, в "нетренированных мозгах", некий "страшный и ужасный, прерывательный вектор", хотя и с внушающим уважение названием, но как-то не слишком "комфортно ложащийся в мозги".
Сие вполне можно сформулировать и на "нормальном языке": условный переход, с помощью аппаратно формируемой команды **call**, в ячейку PC с фиксированным адресом **04h**.
И все дела. И вектор не при чём ("усоп за ненужностью").
Вернемся к нашим "разборкам".
Теперь необходимо выполнить, сформулированные ранее, "правила игры".
В начале ПП прерывания, необходимо сохранить содержимое двух регистров: **Status** и **W**, а в конце, восстановить.
В большинстве случаев, этого оказывается достаточным, но если есть такая необходимость (а она может быть), то можно сохранить/восстановить содержимое и других "жизненноважных" регистров.
Мотивация: после возврата из прерывания, содержимое "жизненноважных" регистров должно быть таким, чтобы работа "основного тела" программы не нарушилась.
То есть, их содержимое должно быть таким же, как и на момент ухода в прерывание.
Для реализации сказанного, в "шапке" программы, нужно "прописать" регистры оперативной памяти со стандартными названиями (рекомендованы разработчиками) **Stat_Temp** и **W_Temp**.
Процедура сохранения и восстановления содержимого регистров **W** и **Status**, в обучающих целях, в тексте программы, отображена в полном объеме, хотя в данном случае, содержимое регистра **W** можно и не сохранять (а соответственно, и не восстанавливать).
И в самом деле, после возврата из ПП прерывания в "основное тело" программы, число, находящееся в регистре **W**, не имеет значения (может быть любым), так как "по его верху", будет записана константа.
Поэтому можно смело аннулировать "прописку" регистра **W_Temp**, а заодно и команды сохранения/восстановления содержимого регистра **W**.

Мало того, то же самое относится и к регистру **Status**.
И в самом деле, уходы в прерывания происходит в 0-м банке, и работа, в ПП прерывания, происходит в нулевом банке (банк не меняется).
К тому же, флаги регистра **Status** в программе не задействованы (их состояния не важны).
Таким образом, в данной программе, вообще можно не применять указанные выше процедуры, но я Вам совсем не советую убирать их "с глаз долой".
От того, что Вы их уберете из текста программы, в ее работе практически ничего не изменится, но при этом Вы потеряете возможность визуального, сознательно-подсознательного запоминания стандартной "конструкции" ПП прерывания, что совсем не есть хорошо.
Если в данной, простенькой программе, без этих процедур обойтись можно, то в других программах они могут, в обязательном порядке, потребоваться.
Смотрим в блок - схему программы.
Необходимо произвести проверку: несущая есть/нет?
Подобными проверками мы уже занимались не один раз, так что повторяться не буду.
Если несущей нет, то осуществляется безусловный переход в ПП выхода из прерывания **EndInt** (о ней - позднее).
Если несущая есть, то необходимо каким-то образом задержать рабочую точку программы в ПП прерывания.
В простейшем случае, можно, после команды **goto EndInt** "поставить" команду **goto CYCLE**, "закольцевав" тем самым рабочую точку программы в "вечном кольце" из 3-х команд (сверху вниз: **btfsc PortB,0, goto EndInt, goto CYCLE**), но в этом случае, проверки нулевого бита регистра **PortB** будут производиться так быстро и так часто, что это может отразиться на качестве работы ПП устройства.
В этом случае, устройство будет сильно подвержено воздействию коротких импульсных помех.
Чтобы увеличить его помехоустойчивость, нужно сделать его "слегка" инерционным, то есть, ввести в его состав некий "фильтр", который "отсеивал бы" значительную часть импульсных помех.
Если речь идет об инерционности, то это означает введение, в состав "вечного кольца", подпрограммы задержки с калиброванным временем задержки (ее величина и определяет инерционность).
При таком "раскладе", подпрограмма задержки (вместе с командами записи констант) будет являться "врезкой" в циклическую ПП **CYCLE**, которая увеличивает время отработки цикла ПП **CYCLE**.
Проще говоря, опрос **0**-го бита регистра **PortB** будет производится не так часто, и с цикличностью, определяемой временем задержки.
В программе **Retr_1.asm**, время отработки цикла ПП **CYCLE** составляет **5,62 мс.** (можно проверить в симуляторе).
Это мой субъективный выбор, можно установить и какое-то другое значение, соответствующим образом изменив значения констант.
Можно также упростить ПП **CYCLE**, заменив 2-хразрядный счетчик на 1-разрядный.
После того, как будет произведено отслеживание и отладка программы в том виде, в котором Вы ее видите, можете, для тренировки, сделать эту замену, а затем произвести соответствующую калибровку времени отработки цикла ПП **CYCLE**.
В тексте программы, найдите ПП **CYCLE** и посмотрите, как сказанное выше, реализуется на практике.
Подпрограммы подобной конструкции мы уже "разбирали" ранее и добавить к этому мне нечего.
При выходе из ПП прерывания (см. ПП **EndInt**), нужно обязательно сбросить флаг прерывания **INT** (**bcf IntCon,1**), восстановить содержимое "жизненноважных" регистров и исполнить команду возврата из прерывания **retfie**.
Пример: если уход в прерывание произошел во время исполнения команды **incfsz SecH,F**, то после возврата из ПП прерывания, рабочая точка программы "встанет" на следующую (после **incfsz SecH,F**) команду (**goto PAUSE**).
После этого, работа в "основном теле" программы будет продолжена.
При отладке программы, Вы увидите как работает стек и сможете проконтролировать его содержимое.

Пояснение: в тексте программы **Retr_1.asm**, Вы не найдете команд переходов в ПП **TRIGGER** (аналогия с ПП **INT**).
Это название я ввел в текст программы для того, чтобы обозначить начало группы команд выбора направлений ретрансляции (для того, чтобы Вы легче "ориентировались" в тексте программы).
Если Вы не испытываете трудностей при работе с текстом программы, то это название (**TRIGGER**) можно удалить. Это никак не отразится на работе программы.
При работе с программой **cus**, сначала была выполнена отладка, а затем, отслеживание.
При работе с программой **Retr_1.asm**, поступим наоборот, то есть "так, как положено".

Отслеживание работы программы в ее "основном теле"

Подготовьтесь к отслеживанию, как это было описано при отслеживании работы программы **cus** (откройте соответствующие окна).
"Врежьте" в "шапку" программы команду **goto INT** (об этом → см. выше).
Сначала работаем в "основном теле" программы.
Заблокируйте команду **goto INT** точкой с запятой.
Команду **goto START** блокировать не нужно.
Сначала "прогоним" рабочую точку программы по тому сценарию ее работы, который определяют настройки **MPLAB** по умолчанию.
В дальнейшем, рекомендую Вам начинать отслеживание работы любой программы именно исходя из этого (лишние "телодвижения" отсутствуют).
Так как были внесены изменения, производим ассемблирование и устанавливаем рабочую точку программы на ее начало.
Пошагово исполняем программу (**F7**).
Команды ПП **START** "проходим без осложнений" ("линейный участок").
ПП **START** программы **Retr_1.asm** построена аналогично ПП **START** программы **cus**, и поэтому останавливаться на ее отслеживании я не буду, сделайте это сами и убедитесь, что, по ходу ее исполнения, все прерывания запрещаются, вывод **RB0** настраивается на работу "на вход", вывод **RB2** настраивается на работу "на выход", включаются подтягивающие резисторы порта B и прерывания **INT** будут происходить по заднему фронту внешнего, управляющего сигнала.
Первой командой ПП **TRIGGER** является бит-ориентированная команда ветвления **btfsc Trigg,0**.
Смотрим в окно **RAM**.
По адресу **0Ch** (адрес регистра **Trigg** в области оперативной памяти, "прописанный" в "шапке" программы), на момент исполнения этой команды, записано число **00h**.
Следовательно, 0-й бит регистра **Trigg** установлен в **0,** и в соответствии с алгоритмом работы команды **btfsc**, следующей после нее, должна исполниться группа команд записи константы **.251** в регистр **PortB**.
Что и имеет место быть (проверьте и убедитесь в этом).
"Стукаем" по **F7**.
"Доходим" до команды **goto Metka_1** и исполняем ее, осуществив безусловный переход на команду, помеченную меткой **Metka_1** (**movlw .197**).
Далее, исполняем процедуру записи констант в регистры **SecH_1** и **SecL_1**, команду "врезки" в ПП задержки **PAUSE_D** (**clrwdt**) и устанавливаем рабочую точку программы на команду **decfsz SecL_1,F**.
Далее, для того чтобы выйти из "закольцовки" ПП **PAUSE_D**, конечно же, можно "постукать" по клавише **F7**, но это займет много времени.
Чтобы быстро "проскочить эту закольцовку", лучше перейти на "автомат".
Назначаем точку остановки на команде **movlw .245**, щелкаем по **зеленому** светофору и ждем окончания процесса.
Рабочая точка "встала" на команду **movlw .245**.
Далее, исполняем все последующие команды и устанавливаем рабочую точку программы на "врезку" ПП **PAUSE** (**clrwdt**).
По ходу этого исполнения убеждаемся, что в регистры **SecH** и **SecL** записались числа **.245** (**F5h**) и **.255** (**FFh**) соответственно, а в регистр **IntCon** записалось число **.144** (**90h**/**10010000**).
По своей конструкции, ПП **PAUSE** такая же, как и ПП **PAUSE_D**.

Значит, назначаем точку остановки на команде **clrf IntCon**, включаем "автомат" и дожидаемся конца процесса.
Исполняем команду **clrf IntCon** и убеждаемся, что содержимое регистра **IntCon** изменилось с **.144** на **.00**.
Запоминаем значение числа, записанного в регистр **Trigg** (в окне **RAM**, адрес **0Ch**), исполняем команду **incf Trigg,F** и оцениваем изменение, произошедшее с содержимым регистра **Trigg**.
Значение записанного в регистр **Trigg** числа, должно увеличиться на **1**.
Исполняем команду **goto START**.
Рабочая точка программы "встанет" на первую команду ПП **START**.
То есть, произойдет ее переход на новый "виток" полного цикла "основного тела" программы, к чему мы и стремились.
Один сценарий работы отслежен. Отслеживаем следующий сценарий.
Ищем первую, после команды **clrf IntCon**, команду ветвления.
Это команда **btfsc Trigg,0**.
Один сценарий ветвления этой команды мы уже отследили. Отслеживаем второй.
Вспоминаем про "уловки".
Заменяем команду **btfsc Trigg,0** на команду **btfss Trigg,0**.
Так как внесены изменения, производим ассемблирование и устанавливаем программу на начало.
По этому сценарию, должен произойти безусловный переход на метку **Metka148**.
Поэтому назначаем точкой остановки команду **movlw .255**.
Запускаем "автомат" и ждем окончания процесса.
Рабочая точка программы "встала" на команду **movlw .255**.
Исполняем эту команду и следующую за ней команду **movlw PortB**.
Убеждаемся, что во все биты регистра **PortB** записались единицы (число **.255**).
Далее, этот сценарий отслеживается точно так же, как и первый сценарий.
А раз это так, то зачем "бестолковкой заниматься"?
В "основном теле" программы, никаких других сценариев работы программы больше нет, следовательно, произведено полное отслеживание.
При этом мы убедились в отсутствии функциональных ошибок, при работе программы в обеих подрежимах сканирования.
Ранее говорилось о том, что количество сценариев работы программы зависит от количества команд ветвления.
В общем виде, это верно, но теперь следует уточнить, каких именно команд ветвления?
В группе команд "основного тела" программы, кроме команды ветвления **btfsc Trigg,0**, Вы увидите еще 4 команды ветвления (**decfsz**, **incfsz**).
Формально, каждая из них "порождает" по 2 сценария, но эти сценарии второстепенны по отношению к рассмотренным выше, так как они "прокручиваются" внутри подпрограмм задержки (**PAUSE_D** и **PAUSE**).
Таким образом, де-юре, эти команды ветвления являются сценарными, а де-факто, сценарии их работы "локализованы" внутри того, что исполняется при любом "раскладе".
Это можно назвать "глобальным сценарием" (или еще как-то).
Таким образом, для того чтобы отследить все "локальные" сценарии, достаточно один раз отследить тот "глобальный" сценарий, в состав которого они входят.

Отслеживание работы программы в пределах ПП прерывания.

Для того чтобы войти в ПП прерывания, в "шапке" программы, необходимо снять блокировку с команды **goto INT** (убрать точку с запятой) и заблокировать команду **goto START** (поставить точку с запятой).
После этого, нужно произвести ассемблирование и установить программу на начало.
При этом, рабочая точка программы "встанет" на команду **goto INT**.
Исполняем эту команду.
Рабочая точка программы "встанет" на первую команду ПП прерывания (**movwf W_Temp**) и далее будет "скакать" по командам ПП прерывания (она будет исполняться).
После исполнения первых 3-х команд, содержимое регистров **Status** и **W** должно скопироваться в регистры **Stat_Temp** и **W_Temp** соответственно.
В данном случае, копироваться будут значения по умолчанию.

В регистре **W**, по умолчанию, установлено число **00h**.
Так как в регистре **W_Temp**, по умолчанию, также установлено число **00h**, то, при сохранении содержимого регистра **W,** в регистре **W_Temp**, в окне **RAM**, никаких изменений не произойдет (кроме **PC**).
В регистре **Status**, по умолчанию, установлено число **18h**.
Проследите (в окнах ***Watch*** и **RAM**), как это число, сначала, запишется в регистр **W** (**movf Status,W**), а затем, скопируется из регистра **W** в регистр **Stat_Temp**.
Пошли дальше.
Перед исполнением команды ветвления **btfsc PortB,0**, необходимо посмотреть на состояние нулевого бита регистра **PortB**.
По умолчанию, он установлен в **0**, следовательно, в соответствии с алгоритмом работы команды **btfsc**, рабочая точка программы "уйдет" сценарий "программа исполняется далее", То есть, "встанет" на команду **movlw .250**.
Так оно и есть.
Далее, исполняем все команды записи констант в регистры **SecH_2** и **SecL_2**.
Убеждаемся, что в регистры **SecH_2** и **SecL_2**, из регистра **W**, скопировались числа **.250** и **.120** соответственно.
Переходим на "врезку" в ПП задержки **PAUSE_2** (**clrwdt**).
Далее, отслеживание работы программы происходит аналогично отслеживанию работы ПП **PAUSE** или **PAUSE_D**.
Собственно говоря, работу ПП **PAUSE_2** можно и не отслеживать, а "пройти ее в автомате", поставив точку остановки на первую команду ПП **CYCLE** (**btfsc PortB,0**).
Так и делаем.
Выставляем точку остановки, щелкаем по кнопке с **зеленым** светофором и убеждаемся, что рабочая точка программы "встала" на команду **btfsc PortB,0**.
Это свидетельствует о том, что внутренний цикл подпрограммы прерывания успешно пройден ("глюков/мин" нет).
Для того чтобы в этом окончательно убедиться (ну и в обучающих целях тоже), не снимая установленную точку остановки, можно еще раз (да и сколько угодно раз) щелкнуть по кнопке с **зеленым** светофором и капитально удостовериться в том, что после отработки "автомата", рабочая точка программы снова окажется на команде **btfsc PortB,0.**
Это свидетельствует о том, что при наличии несущей, рабочая точка программы "железобетонно уйдет в вечное кольцо" ПП **CYCLE**), в чем и требовалось радостно убедиться.
Один сценарий отслежен. Отслеживаем следующий.
"Встаем" все на ту же команду **btfsc PortB,0**.
Применяем "уловку", заменив эту команду на команду **btfss PortB,0**.
Производим ассемблирование и устанавливаем программу на начало.
Так как должен произойти безусловный переход на первую команду ПП **EndInt**, то устанавливаем точку остановки на команде **bcf IntCon,1**.
Запускаем "автомат" и ждем окончания его отработки.
Рабочая точка программы "встала" на команду **bcf IntCon,1**.
Исполняем ее и убеждаемся, что 1-й бит регистра **IntCon** установился в **0** (флаг прерывания **INT** программно сброшен).
Далее, исполняем, рекомендованный разработчиками, стандартный "набор" из 4-х команд восстановления содержимого регистров **Status** и **W** (результат можно проконтролировать).
Рабочая точка программы "встала" на команду **retfie**.
Исполняем команду **retfie** и получаем предупреждение (**MPLAB** "в панике") о том, что **стек пуст** и соответственно, извлекать из него нечего. Хоть "шаром покати".
Но паниковать не нужно. В конце-концов, мы кто? Гомо сапиенсы или мартышки?
Щелкнув по ***OK***, спокойно закрываем окно предупреждения, после чего рабочая точка программы устанавливается на начало (в данном случае, на команде **goto INT** в "шапке" программы).
Поза мыслителя.
Вопрос: "В чем дело? Почему стек пустой"?
Ответ: переход в ПП прерывания был осуществлен не из "зоны" разрешения прерываний "основного тела" программы, а при помощи "уловки", то есть, проще говоря, условного перехода (см. **"виртуальный" call**) не было, а следовательно и в стек ничего не записалось.
"С другого бока": так как, по ходу отслеживания, никаких команд условных переходов не

исполнялось, а стек, по умолчанию, пуст, значит и на момент исполнения команды **retfie**, он также будет пуст, о чем мы и получили добросовестное предупреждение.
А раз это так, то при щелчке по ***OK***, **MPLAB** "принудительно" устанавливает рабочую точку программы на начало, как бы предлагая найти и устранить ошибку, а затем попробовать повторить отслеживание еще раз.
Таким образом, для "полноценного" отслеживания данного сценария работы ПП прерывания, необходимо "уйти" в прерывание из "зоны" разрешения прерываний "основного тела" программы.
Проще всего это сделать, не задействуя функции стимула, а при помощи очередной "уловки", смысл которой заключается в следующем.
В списке команд, Вы не обнаружите команды условного перехода в ПП прерывания, в нем имеется только команда возврата из ПП прерывания (**retfie**).
Но это вовсе не означает, что "уход" в ПП прерывания происходит по какому-то "таинственному указанию святого духа".
Эта команда вполне конкретна и называется она **call**
(выше, я ее называл **"виртуальный" call**).
Просто она исполняется не программно, а аппаратно. По факту возникновения события прерывания любого типа.
Таким образом, чтобы проверить/сымитировать уход в прерывание из "зоны" разрешения прерываний "основного тела" программы, необходимо просто-напросто "врезать" в нее команду **call INT**.
Для того чтобы быстро "добраться" до этой команды (не ждать окончания отработки счетчика), ее лучше "врезать" между командами **clrwdt** и **decfsz SecL,F**.
Для работы в "основном теле" программы, также необходимо заблокировать команду **goto INT** и разблокировать команду **goto START**.
Делаем это.
Производим ассемблирование, и после получения сообщения о безошибочном ассемблировании, устанавливаем программу на начало (**goto START**), назначаем точку остановки на команде **call INT** и запускаем "автомат".
Ждем его отработки, после чего рабочая точка программы устанавливается на команде **call INT** (то есть, таким образом, мы "добрались" до этой команды).
А теперь разбираемся со стеком.
Сначала нужно открыть специальное окно, в котором мы будем наблюдать содержимое стека.
Это делается так: в главном меню **MPLAB**, щелкаем по слову ***Window*** и в выпадающем списке, щелкаем по строке ***Stack***.
Откроется окно ***Stack Window***.
В нем пока ничего нет (т.к. команд условных переходов не было).
Расположите это окно в любом удобном для Вас месте обычным, для "Виндов", способом.
Исполняем команду **call INT**.
Рабочая точка программы "уйдет" в ПП прерывания и "встанет" на первую ее команду (**movwf W_Temp**).
А теперь посмотрите в окно стека.
В нем появилась строка с адресом той команды, на которую, после возврата рабочей точки программы из ПП прерываний, она "встанет" (с адресом возврата).
Этот адрес - **003Bh**.
Откройте окно **ROM** и убедитесь в том, что команда с этим адресом (**decfsz SecL,F**) является следующей, после команды **call INT**.
При переходе в ПП прерывания по активному перепаду на входе **RB0/INT**, по сути, происходит то же самое.
Разница только в том, что положение команды **call INT**, в тексте программы, фиксировано, а **"виртуальный" call** может сформироваться сразу же после отработки любой из команд "зоны" разрешения прерываний.
Напоминаю Вам, что ранее мы отслеживали исполнение 2-го сценария работы ПП прерываний.
Еще раз "прогоним" по нему рабочую точку программы, но на этот раз, мы "вошли" в ПП прерываний "как положено", то есть, из "зоны" разрешения прерываний "основного тела" программы (в вершину стека "заложен" адрес возврата).

Так как при первом "прогоне", мы уже отследили работу этого сценария и убедились, что все в порядке, то назначаем точкой остановки команду **retfie** и запускаем "автомат".
После его отработки, рабочая точка программы "встала" на команду **retfie**.
Исполняем ее.
Вершина стека очистится (стек станет опять пустым), рабочая точка программы "вернется" в "основное тело" программы и установится на команде **decfsz SecL,F**, в чем и требовалось убедиться.
Итак, мы разобрались со всеми сценариями работы программы и убедились в отсутствии ошибок функционального характера, а заодно и "заглянули" в стек.
Программа отслежена.
После всех этих "катаклизмов", нужно "навести в программе порядок", вернув ее к исходному состоянию (убрать все "уловки").

Отладка программы

В результате отслеживания работы программы, мы убедились в том, что программа работает в соответствии с задуманным алгоритмом ее работы.
При этом, внимания на ее временнЫе характеристики не обращалось.
Так как в большинстве программ, имеются калиброванные, по времени (с той или иной точностью, это зависит от специфики разрабатываемого устройства), задержки, то в первую очередь, именно к коррекции величин этих задержек (под задуманные) и сводится процесс отладки программы.
Обращаю Ваше внимание на то, что отлаживаться может как "локальная" ПП задержки, так и более "навороченная" ПП задержки, имеющая внутри своего цикла несколько "локальных" ПП задержек.
Примером "локальной" ПП задержки может послужить, например, ПП **PAUSE_2**, а примером "навороченной" ПП задержки может послужить ПП **CYCLE** (см. текст программы **Retr_1.asm**).
И ту, и другую можно отладить, то есть, изменить времязадающие константы (а при необходимости, добавить или удалить калибровочные **NOP**ы) таким образом, чтобы время отработки задержек было равно расчетным значениям.
Для того чтобы это сделать без ошибок, нужно знать, как программа работает.
В процессе отладки, программист должен уметь "вычленять", из текста программы, оба типа ПП задержек (см. выше) и четко представлять себе функции этих задержек в "контексте" всей программы.
Только после этого имеется гарантия того, что он поставит точки остановок точно и без ошибок.
Разбираемся с задержками программы **Retr_1.asm**.
Ранее, было сформулировано требование к величине защитного интервала времени: он должен быть равен **примерно 60 мс.**
Определяемся с "границами" защитного интервала времени.
Он начинает формироваться после исполнения команды **movwf PortB** (таких команд две), как для 1-го, так и для 2-го сценария работы программы в "основном теле" программы.
Сразу "настораживаемся": два сценария могут исполняться за разное время, а нам, предположим, нужно сделать так, чтобы оба сценария выполнялись за одинаковое время.
В учебно-тренировочных целях, производим оценку этой разницы.
По вполне понятным причинам, общие команды этих 2-х сценариев в расчет не берутся.
Остаются те части этих сценариев, которые не являются общими.
В них, рабочая точка программы, в зависимости от сценария ее исполнения, "движется" по-разному.
Соответственно, интервалы времени их отработки тоже могут быть разными.
Вот Вам и разница во времени.
Давайте разбираться.
После команды **btfsc Trigg,0**, происходит "разветвление" сценариев, а "слияние" сценариев происходит на команде **movlw .197** (на это намекалось выше).
Для того чтобы **выровнять** время исполнения этих сценариев, необходимо подсчитать количество машинных циклов, за которые они исполняются.
Если при этом выявятся различия, то необходимо произвести их выравнивание калибровочными **NOP**ами.
В данном случае, практической необходимости в "выравнивании" сценариев нет, но

существуют и программы, необходимым условием "полноценной" работы которых является одинаковое время исполнения двух или более ее "локальных" сценариев, "расходящихся из одной точки", а затем "сходящихся в одну точку" (например, программы, созданные под точные, измерительные приборы).
Раз уж "подвернулся такой повод", то, как говорится, "грех его упускать".
Поэтому, в учебно-тренировочных целях, ниже, будет произведено выравнивание "локальных" сценариев.
Итак, "расхождение сценариев из одной точки" происходит от команды **btfsc Trigg,0**.
"Схождение сценариев в одну точку" происходит на команде **movlw .197**.
Выставляем точку остановки на команде **btfsc Trigg,0**.
Сбрасываем программу на начало и "доходим", до этой точки остановки, в "автомате".
Проблем при этом не будет, так как установки MPLAB по умолчанию, позволяют это сделать (в противном случае, пришлось бы применять "уловку").
Выставляем точку остановки на команде **movlw .197**.
Сбрасываем программу на начало, вызываем секундомер, сбрасываем секундомер в ноль, жмем на кнопку с **зеленым** светофором.
Секундомер показал **6 м.ц.**
Чтобы отработать 2-й сценарий, нужно применить "уловку", в виде замены команды **btfsc Trigg,0** на команду **btfss Trigg,0**.
Ассемблируем, выставляем те же точки остановки и т.д. (те же действия).
Секундомер показал **5 м.ц.**
"Разнобой" в **1 м.ц.**
Для его устранения, нужно "врезать" один **nop** после команды **movwf PortB** (той, которая следует за командой **movlw .255**).
Выравнивание сценариев произведено (оба будут исполняться за **6 м.ц.**).
В тексте программы, этого **NOP**а нет по причине того, что "разнобой" в 1 м.ц. никакой "погоды не делает" но, при желании, Вы его можете "врезать".
Если Вы добавили в текст программы **nop**, то нужно произвести ассемблирование.
Этот пример простейшего выравнивания времени исполнения сценариев приведен для того, чтобы было понятно, о чем идет речь.
В других программах, такое выравнивание может быть архинеобходимым (об этом пойдет речь в одном из следующих разделов), так что обратите на это внимание.
Например, если в частотомере не выровнять сценарии, то его показания не будут стабильными даже при абсолютно стабильной, измеряемой частоте.
После того, как мы убедились, что оба сценария будут исполняться за одно и то же время (если имеется такая необходимость), можно заняться калибровкой (но не наоборот).
Сбросьте программу **Retr_1.asm** на начало (на **goto START**) и, в пошаговом режиме (а можно и в "автомате"), "встаньте" на команду **movwf PortB** (на ту, которая находится после команды **movlw .251**).
Вызовите секундомер и сбросьте его на ноль.
Назначьте точку остановки на команде **movlw .245** (конец формирования защитного интервала времени) и запустите "автомат".
После того, как он отработает, секундомер покажет ровно **60 мс.**
Естественно, что в этом случае, числовые значения констант защитного интервала времени уже подобраны **(.197** и **.121**).
Можете, для тренировки, поизменять их и посмотреть (по секундомеру), что из этого получится.
Процесс отладки подобного рода ПП задержки уже был подробно расписан выше, так что повторяться не буду.
Замечание: фактически, формирование защитного интервала времени заканчивается не после выхода рабочей точки программы из ПП **PAUSE_D**, а после исполнения команды **movwf IntCon**, то есть, величина защитного интервала времени окажется на 6 мкс. (6 команд по одному м.ц.) больше (60 006 мкс.), что с учетом допуска в "плюс-минус 2 лаптя", вполне приемлемо.
Отладка остальных ПП задержек - аналогична.
"Пройдитесь" по этим ПП задержек самостоятельно. После этого Вы, должны получить следующие результаты:
1. "Ширина зоны" разрешения прерываний: **11,28 мс.** ("границы": верхняя - **movwf IntCon**, нижняя - **clrf IntCon**).

2. В ПП прерывания, интервал времени отработки одного цикла задержки: **5,62 мс.** (точка остановки одна: **btfsc PortB,0** (от этой точки остановки, и через один "виток", до нее же).
Примечание: собственно говоря, отладки, как таковой, и не было (константы не менялись), а была просто проверка величин задержек, которые уже были отлажены ранее.
Чтобы процесс отладки был "полноценным", для тренировки, можете изменить исходные требования к этим величинам, и руководствуясь принципом отладки, описанным при отладке программы **cus**, произвести отладку программы **Retr_1.asm**.
Итак, отладка закончена, и HEX-файл программы **Retr_1.asm** можно открывать в программе, обслуживающей Ваш программатор, после чего можно "прошить" ПИК.
При этом, вероятность того, что устройство будет работать именно так, как задумано, велика.
А теперь о программе, выполняющей то же самое, но без уходов в прерывания.

Файл текста программы называется Retr_3.asm (находится в папке ***"Тексты программ"***).

Программа выглядит так:

```
;*******************************************************************************
; Retr_3.asm                    ВАРИАНТ БЕЗ ПРЕРЫВАНИЙ
;                          Сканер для ретранслятора VERTEX-7000VXR.
; Автор: Корабельников Евгений Александрович  г.Липецк  декабрь 2004г.
; E-mail: karabea@lipetsk.ru                 http://ikarab.narod.ru
;*******************************************************************************
; Позволяет перевести VERTEX-7000VXR и другие подобные ретрансляторы из режима
; односторонней ретрансляции в режим двухсторонней ретрансляции без потери
; качества работы.
; Объем программы: 40 слов в памяти программ.
; Используется микроконтроллер PIC16F84A. Частота кварца 4000кГц.
;*******************************************************************************
            LIST      p=16F84a      ; Используется PIC16F84A.
            __CONFIG  03FF5H        ; WDT включен, бит защиты не установлен.
;===============================================================================
; Определение  положения регистров специального назначения.
;===============================================================================
OptionR     equ         01h         ; Option - банк1
Status      equ         03h         ; Регистр Status
PortB       equ         06h         ; Порт B
TrisB       equ         06h         ; Tris B - Банк1
IntCon      equ         0Bh         ; Регистр IntCon
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
Trigg       equ         0Ch         ; Переключатель направления ретрансляции.
SecH        equ         1Eh         ; Старший байт счетчика времени сканирования.
SecL        equ         1Fh         ; Младший байт счетчика времени сканирования.
SecH_1      equ         1Ch         ; Старший байт счетчика защитного интервала
                                    ; времени.
SecL_1      equ         1Dh         ; Младший байт счетчика защитного интервала
                                    ; времени.
;===============================================================================
; Определение места размещения результатов операций.
;===============================================================================
F           equ         1           ; Результат направить в регистр.
;===============================================================================
; Определение положения бита выбора банка.
;===============================================================================
RP0         equ         5           ; Бит выбора банка.
;===============================================================================
; Точка входа в программу.
;===============================================================================
            org         0           ; Начать выполнение программы с нулевого
            goto        START       ; адреса PC
;*******************************************************************************
```

```
;*******************************************************************************
;                              РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;*******************************************************************************
; Подготовительные операции.
;-------------------------------------------------------------------------------
START       clrf        IntCon      ; Запретить все прерывания.
            clrwdt                  ; Сбросить сторожевой таймер WDT.
            bsf         Status,RP0  ; Установить банк 1.
            movlw       .1          ; RB0 работает на вход,
            movwf       TrisB       ; остальные - на выход.
            movlw       .0          ; Включить подтягивающие резисторы порта B.
            movwf       OptionR     ; Остальное - не существенно.
            bcf         Status,RP0  ; Установить банк 0.
;-------------------------------------------------------------------------------
; Выбор направления ретрансляции.
;-------------------------------------------------------------------------------
TRIGGER     btfsc       Trigg,0     ; Проверка значения нулевого бита
                                    ; регистра Trigg.
            goto        Metka148    ; Если это значение =1, то переход
                                    ; на метку Metka148.
                                    ; Если это значение =0, то программа
                                    ; исполняется далее.
            movlw       .251        ; Закладка константы .251 (1111 1011)
                                    ; в регистр W.
            movwf       PortB       ; Копирование .251 из регистра W
                                    ; в регистр PortB (выбор прямого направления
                                    ; ретрансляции: ПРМ-X, ПРД-Y).
            goto        Metka_1     ; Безусловный переход на метку Metka_1.
Metka148    movlw       .255        ; Закладка константы .255 (1111 1111)
                                    ; в регистр W.
            movwf       PortB       ; Копирование .255 из регистра W
                                    ; в регистр PortB (выбор обратного направления
                                    ; ретрансляции: ПРМ-Y, ПРД-X).
;-------------------------------------------------------------------------------
; Формирование защитного интервала времени (ожидание окончания переходных
; процессов, возникающих при смене направлений ретрансляции) равного,
; примерно, 60 мс.
;-------------------------------------------------------------------------------
Metka_1     movlw       .197        ; Закладка в регистр W константы .197
            movwf       SecH_1      ; Копирование константы .197 из регистра W
                                    ; в регистр SecH_1.
            movlw       .121        ; Закладка в регистр W константы .121
            movwf       SecL_1      ; Копирование константы .121 из регистра W
                                    ; в регистр SecL_1.
PAUSE_D     clrwdt                  ; Сброс WDT.
            decfsz      SecL_1,F    ; Декремент содержимого младшего разряда
                                    ; счетчика SecL_1.
            goto        PAUSE_D     ; Если результат декремента не=0, то переход
                                    ; в ПП PAUSE_D.
            incfsz      SecH_1,F    ; Если результат декремента =0, то инкремент
                                    ; старшего разряда счетчика SecH_1.
            goto        PAUSE_D     ; Если результат инкремента не=0, то переход
                                    ; в ПП PAUSE_D.
                                    ; Если результат инкремента =0, то программа
                                    ; исполняется далее.
;===============================================================================
; Формирование интервала времени сканирования.
;===============================================================================
            movlw       .245        ; Закладка константы .245 в регистр W.
            movwf       SecH        ; Копирование .245 из регистра W
                                    ; в регистр SecH.
            movlw       .255        ; Закладка константы .255 в регистр W.
            movwf       SecL        ; Копирование .255 из регистра W
                                    ; в регистр SecL.
```

```
;-------------------------------------------------------------------------------
PAUSE       clrwdt                  ; Сброс WDT.
            btfss       PortB,0     ; Несущая есть?
            goto        PAUSE       ; Да.
            btfss       PortB,0     ; Нет. Программа выполняется далее.
                                    ; Несущая есть ?
            goto        PAUSE       ; Да.
            decfsz      SecL,F      ; Нет. Декремент содержимого младшего разряда
                                    ; счетчика SecL.
            goto        PAUSE       ; Если результат декремента не=0, то переход
                                    ; в ПП PAUSE.
            incfsz      SecH,F      ; Если результат декремента =0, то инкремент
                                    ; старшего разряда счетчика SecH.
            goto        PAUSE       ; Если результат инкремента не=0, то переход
                                    ; в ПП PAUSE.
                                    ; Если результат инкремента =0, то программа
                                    ; исполняется далее.
;-------------------------------------------------------------------------------
; Изменение направления ретрансляции.
;-------------------------------------------------------------------------------
            incf        Trigg,F     ; Увеличение на 1 (инкремент)содержимого
                                    ; переключателя направления ретрансляции.
            goto        START       ; Переход на новый цикл сканирования.
;*******************************************************************************
            end                     ; Конец программы.
```

Сравните тексты программ **Retr_1.asm** и **Retr_3.asm**
Если Вы хорошо усвоили предыдущую информацию, то текст программы **Retr_3.asm** , а также и его отличия от текста программы **Retr_1.asm**, не будут вызывать у Вас затруднений. Если это не так, то вернитесь назад и найдите там ответы на Ваши вопросы.
Программу **Retr_3.asm** можно использовать как некую "лакмусовую бумажку": пока Вам в ней что-то не понятно, то не стОит переходить к изучению следующих разделов, а лучше разобраться с предыдущими.
В идеале, программа **Retr_3.asm** должна быть Вам понятна "от и до".
Замечания по прерываниям.
Целью "вышележащих разборок" с прерываниями является только формирование общих понятий о них.
Описанное выше устройство не идеально в том смысле, что имеется достаточно большая "зона" запрета прерываний.
Хотя, в данном случае, специфика работы ретранслятора Vertex-7000VXR такова, что допускает наличие такой "зоны", но при создании программ под другие устройства, такой подход может быть неприемлем.
Более детальные "разборки" с прерываниями будут произведены позднее.

Дополнительно

Разберемся с той "мутью", которая подпортила нервы тем, кто ее заметил (5 баллов за внимательность) и выдал сигналы SOS.
Открываем проект **Retr_1**.
При этом, **MPLAB** выставит совершенно однозначные настройки по умолчанию.
Нас интересует регистр **PortB**, и в частности, его бит с **№ 0**.
По умолчанию, он установлен в **0** (см. окно **SFR**).
Пошагово исполняем программу до момента установки единицы в бите **№0** регистра **TrisB** (перевод вывода **RB0** на работу "на вход").
После этого смотрим, не изменилось ли состояние нулевого бита регистра **PortB**?
Не изменилось. Как был **0**, так он и остался.
Вывод: на выводе порта, настроенном на работу как "на выход", так и "на вход", по умолчанию, **MPLAB** выставляет **0**.
Пошагово исполняем программу далее, обращая внимание только на то, что связано с командами, которые обращаются к содержимому регистра **PortB**.

По ходу исполнения программы, первой такой командой является команда копирования содержимого регистра **W** в регистр **PortB** (**movwf PortB**).
В аккумуляторе (**W**) находится число **.251**, и казалось бы, именно оно должно скопироваться в порт B.
Исполняем команду **movwf PortB**.
Смотрим в окно **SFR**.
А скопировалось-то не **.251**, а **.250**.
???
Ради "спортивного интереса", можете поподставлять, вместо **.251**, другие числовые значения констант и убедиться в том, что все нечетные числа "превращаются" в четные, а с четными числами проблем нет.
О чем это говорит?
Это говорит о том, что в бите **№0** регистра **PortB**, "намертво застолблён" ноль, выставленный **MPLAB** по умолчанию.
Дело в том, что **MPLAB отражает состояния выводов порта**.
То есть, если вывод порта настроен на работу "на выход", то по причине того, что в этом случае, выход защелки подключается к выводу порта, состояние вывода порта будет соответствовать состоянию его "персональной" защелки.
Что же тогда мы видим в бите **№0**?
Мы видим уровень, выставленный **MPLAB** по умолчанию, **для вывода порта работающего "на вход"** (**0**). Вот и весь сказ.
Свидетельствует ли это о какой-то ошибке в работе программы/"железяки"?
Нет, не свидетельствует.
Образно выражаясь, откуда **MPLAB**у знать, какой уровень выставит внешнее устройство?
Вот он, не мудрствуя лукаво, по умолчанию, и выставляет на тех выводах порта, которые работают "на вход", нулевые уровни.
С помощью функций стимула, на выводе порта, который работает "на вход", можно сымитировать единицу (в **MPLAB** это можно сделать), но в данном случае, проще узнать "в чем собака порылась" и сделать соответствующие выводы, чем что-то имитировать.
От этого "извилина крепчает" и прибавляется опыт.

12. Организация вычисляемого перехода. Работа с EEPROM памятью данных.

В предыдущих разделах я стремился объяснить Вам некоторые фундаментальные основы программирования микроконтроллеров (любых).
На данный момент Вы должны отчетливо представлять себе общие принципы составления текста программы, движения рабочей точки программы по командам текста программы, и самое главное, Вы должны понять зачем нужны подпрограммы задержек, их предназначение и основные функции, так как программирование без задержек это все-равно что машина без мотора.
Все циклические подпрограммы "базируются" на задержках и такой термин, как "закольцовка", тоже "из этой же оперы".
Именно задержки составляют основу программы.
И сама программа представляет собой одно большое "кольцо", внутри которого может быть какое-то количество "колец рангом пониже", а в этих "кольцах", в свою очередь, может находиться какое-то количество "колец рангом еще ниже" и т.д.
Движение рабочей точки программы по полному циклу программы можно сравнить со взлетом и посадкой самолета, между которыми он совершает несколько "мертвых петель".
В идеале, программист должен четко и ясно представлять себе функцию каждого кольца, его "конструкцию" и правила его функционирования.
Есть еще и другие, несомненно, важные "вещи", на которые, до конца не разобравшись с "кольцевой природой" программ, "клюют" начинающие программисты, совершая тем самым серьезную, стратегическую ошибку.
Это вполне может привести к трудностям в представлении общей "картины" работы программы, даже если человек неплохо разбирается с ее составными частями.
Так что не жалейте времени на "разборки" с задержками ("кольцами").
Этот труд однозначно окупится.
И еще один добрый совет: если Вы хотите разобраться с работой программы, то не пожалейте времени на ее общий анализ и составление, по его результатам, блок-схемы программы. В дальнейшем, это сэкономит много времени и нервов.
А теперь начинаем разбираться с другими важными "вещами".

Организация вычисляемого перехода

Понятие вычисляемого перехода было вкратце сформулировано во 2-м разделе.
Теперь, подробнее.
Нам уже известно, что команды ветвления "порождают" 2 сценария работы программы.
А как быть, если нужно "компактно разветвиться", например, на 3, 4, 5 и более сценариев?
Это можно сделать при помощи **вычисляемого перехода** (стандартное название операции).
Вычисляемый переход осуществляется при помощи команды **addwf PC,F**, которая формально описывается так: сложить содержимое регистров **W** и **PC**, с сохранением результата сложения в регистре **PC** (имеется ввиду младший разряд счетчика команд с названием **PCL**).
Неформально, эта команда описывается так: увеличить (прирастить) текущее значение содержимого счетчика команд **PC** на величину содержимого регистра **W**.
То же самое можно сказать и о формальном толковании команды **addwf PC,W**.
Разница только в том, что результат сложения будет сохранен в регистре **W**.
Таким образом, из-за того, что результат операции сохранится в регистре **W**, будет осуществлено приращение не счетчика команд, а аккумулятора.
По этой причине, команду **addwf PC,W** нельзя использовать в качестве команды вычисляемого перехода.
Итак, для организации вычисляемого перехода, применяется команда **addwf PC,F**
Эта команда, в тексте программы, может располагаться где угодно (там, куда ее вставит программист), и соответственно, адрес этой команды, в памяти программ, может быть различным.
Так вот, для вычисляемого перехода, этот адрес является как бы начальной точкой отсчета.
Пример: программист составляет программу. Он "дошел до того ее места", где нужно "разветвить" программу, например, на 4 сценария (для того чтобы впоследствии "уйти" на исполнение того или иного из этих 4-х сценариев).

Можно организовать серию последовательных проверок, а можно организовать и вычисляемый переход.
В последнем случае, можно обойтись количеством команд меньшим, чем в первом случае, и "в мозги это ложится" более комфортно.
Такого рода выигрыш тем заметнее, чем на большее количество сценариев нужно "разветвиться".
Прежде чем вставить в текст программы команду **addwf PC,F**, необходимо определиться с критерием перехода, то есть, с содержимым чего-то, что определяет, какой именно из 4-х сценариев будет далее исполняться.
Это "чего-то" есть регистр **W**, а критерием перехода является его содержимое, то есть, число, находящееся в регистре **W** на момент исполнения команды **addwf PC,F**.
Это число и есть **приращение** счетчика команд **PC**.
Еще раз повторяю, что команда **addwf PC,F**, в памяти программ, может иметь любой адрес, и в случае организации вычисляемого перехода, этот адрес определяет только положение этой команды в памяти программ и не влияет на выбор дальнейших сценариев работы программы.
Этот выбор зависит от **приращения** счетчика команд **PC**.
Какие числа должны быть записаны в регистр **W**?
Для того чтобы процедура вычисляемого перехода получилась "компактной", значения этих чисел, в данном случае (4 сценария), должны быть равны 0, 1, 2, 3 (почему именно такие значения, а не 1, 2, 3, 4, Вы поймете позже).
Если сценариев большее чем 4, то этот числовой ряд нужно продолжить (количество этих чисел должно быть равно количеству сценариев).
После осуществления одного из нескольких возможных приращений содержимого счетчика команд **PC**, будет осуществлен переход (стек не задействован) на одну из команд начала исполнения сценариев программы, которая и будет исполнена в последующих машинных циклах.
Командами начала исполнения сценариев могут быть:

- ***Команды* goto**.
  Последним сценарием, может быть (а может и не быть. Зависит от задумки) ***сценарий "программа исполняется далее".***
  Такой вычисляемый переход применяется в тех случаях, когда выбор того или иного сценария работы программы нужно поставить в зависимость от внешнего управления (например, от состояния клавиатуры) ***или от результатов работы программы*** (например, от результата вычисления).
- ***Команды* retlw**.
  Такой вычисляемый переход применяется для выборок данных из таблицы данных (в том числе и для перекодировок)**.**

Конечно же, можно как угодно, "некомпактно разбросать", по тексту программы, команды начала исполнения сценариев, но зачем нужны лишние проблемы?.
"Компактной группой" я называю группу команд начала исполнения сценариев программы, располагающуюся сразу же после команды **addwf PC,F**.
То есть, адреса этих команд, в памяти программ, "идут друг за другом" (команды не "разбросаны" по памяти программ).
Например, для случая 4-сценарной работы, перед исполнением команды **addwf PC,F**, нужно откуда-то скопировать, в регистр **W**, одно из чисел, находящихся в числовом диапазоне от нуля до трех включительно.
Если это сделать, то после исполнения команды **addwf PC,F**, произойдет переход ("прыжок") рабочей точки программы на одну из четырех команд начала исполнения сценариев программы (после условных и безусловных, это третья разновидность переходов).
В дальнейшем, после перехода на начало исполнения выбранного сценария, он и будет исполнен.
Вопрос: "Откуда, в регистр **W**, могут быть записаны числа, значения которых определяют дальнейший выбор того или иного сценария работы программы"?
Ответ: из регистра общего назначения, предназначенного для осуществления этой операции. Соответственно, на момент начала копирования, содержимое этого регистра общего назначения должно соответствовать, сформулированному выше, "условию компактности".

Если "привязаться" к случаю управления сценариями работы программы, с помощью клавиатуры, реализующей одно из 4-х возможных состояний, то для этого потребуется 2 кнопки.
В конечном итоге, содержимое регистра **W** формируется по результатам опроса клавиатуры.
Можно опросить клавиатуру и из большего количества кнопок.
В этом случае, количество возможных состояний увеличится и можно реализовать большее количество сценариев работы программы.
В любом случае, "прыжок" рабочей точки программы на одну из команд начала исполнения сценариев программы, происходит по принципу: ***если в регистре W, на момент исполнения команды addwf PC,F, записано число N, то после исполнении команды addwf PC,F, произойдет переход рабочей точки программы на команду начала исполнения сценария программы, с номером N+1.***
Обращаю Ваше внимание на то, что этот "прыжок" (переход) происходит без применения команды перехода, а **за счет приращения содержимого счетчика команд PC**, которое происходит после исполнения команды **addwf PC,F**.
Итак, ближе к практике.

Файл с примерами вычисляемых переходов addwfpc.asm
(находится в папке ***"Тексты программ"***).

Это выглядит так:

```
;*******************************************************************************
;                     ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ВЫЧИСЛЯЕМЫХ ПЕРЕХОДОВ
;                                  ПРИМЕР № 1
;*******************************************************************************
; В данном случае, клавиатура подключена к выводам RA0 и RA1
; (2 кнопки или тумблера).
; В "шапке" программы должен быть "прописан" регистр PC (PC  equ  02h) и до
; исполнения команды movf PortA,W, выводы порта А, подключенные к клавиатуре,
; должны быть установлены на работу "на вход".
;-------------------------------------------------------------------------------
            ......      ......      ;
            ......      ......      ;
            movf        PortA,W     ; Опрос клавиатуры (копирование числа,
                                    ; присутствующего на выводах порта А,
                                    ; в регистр W).
;-------------------------------------------------------------------------------
;                       Применение вычисляемого перехода.
; (переход на исполнение различных сценариев в зависимости от
; состояния клавиатуры)
;-------------------------------------------------------------------------------
            addwf       PC,F        ; Увеличение PC в зависимости от состояния
                                    ; клавиатуры (числа, записанного при опросе
                                    ; клавиатуры, в регистр W).
            goto        Function    ; Режим установки ПЧ.               0000 0000
            goto        Plus        ; "Плюс ПЧ".                        0000 0001
            goto        Minus       ; "Минус ПЧ".                       0000 0010
            ......      ......      ; Программа исполняется далее       0000 0011
            ......      ......      ;

;===============================================================================
;                                  ПРИМЕР № 2
;===============================================================================
; В "шапке" программы должен быть "прописан" регистр PC (PC  equ  02h).
;-------------------------------------------------------------------------------
            ......      ......      ;
            ......      ......      ;
            call        TABLE       ; Переход по стеку в ПП TABLE.

            ......      ......      ;<--Возврат по стеку из ПП TABLE, после чего
            ......      ......      ; программа исполняется далее.
;-------------------------------------------------------------------------------
```

```
;                   Применение вычисляемого перехода.
; (преобразование двоично-десятичного кода в код 7-сегментного индикатора)
;-------------------------------------------------------------------------------
TABLE       addwf       PC,F            ; Содержимое счетчика команд PC увеличивается
                                        ; на величину содержимого аккумулятора W.
            retlw       b'00111111'     ; ..FEDCBA = 0
            retlw       b'00000110'     ; .....CB. = 1
            retlw       b'01011011'     ; .G.ED.BA = 2
            retlw       b'01001111'     ; .G..DCBA = 3
            retlw       b'01100110'     ; .GF..CB. = 4
            retlw       b'01101101'     ; .GF.DC.A = 5
            retlw       b'01111101'     ; .GFEDC.A = 6
            retlw       b'00000111'     ; .....CBA = 7
            retlw       b'01111111'     ; .GFEDCBA = 8
            retlw       b'01101111'     ; .GF.DCBA = 9
            retlw       .00             ; ........ = Все секторы погашены.
            retlw       .113            ; .GFE...A = F  Признак режима установки ПЧ.
            retlw       .246            ; HGFE.CB. = H. Признак режима "плюс ПЧ".
            retlw       .184            ; H.FED... = L. Признак режима "минус ПЧ".
            retlw       .241            ; HGFE...A = F. Признак режима частотомера.
;-------------------------------------------------------------------------------
```

Обращаю Ваше внимание на то, что этот файл, формально, не является программой (он является частью программы и не оформлен как "полноценная" программа) и проекта под него создавать не нужно.
Открыть его в **MPLAB** можно как обычный файл (***File → Open***).

Рассмотрим **пример № 1.**
Этот пример иллюстрирует принцип "разветвления" программы на 4 сценария, выбор одного из которых ставится в зависимость от текущего состояния клавиатуры, состоящей из двух кнопок, подключенных между выводами **RA0**, **RA1** порта А и корпусом.
Для того чтобы, на выводах **RA0** и **RA1**, обеспечить единичные уровни при отжатых кнопках (кнопки - с нормально разомкнутыми контактами. Порт А не имеет внутренней "подтяжки"), необходимо их "подтянуть", к цепи +5v, внешними, подтягивающими резисторами.
В соответствии с изложенным выше принципом, команды начала исполнения сценариев программы должны располагаться компактно, сразу же после команды **addwf PC,F**, что Вы и видите.
В данном случае, вычисляемый переход на исполнение одного из 4-х сценариев программы, производится следующим образом.
С помощью команды **movf PortA,W**, производится опрос клавиатуры.
Так как команда **movf** является байт-ориентированной, то в регистр **W** записывается байт (8 битов).
Так как задействован порт А, имеющий 5 выводов (PIC16F84A), то в битах № 5, 6, 7 регистра **PortA**, по умолчанию, всегда будут выставляться нули.
Таким образом, при наличии нулей на выводах **RA2, RA3, RA4** (это нужно обеспечить до "влёта" в эту процедуру), возможные числовые значения содержимого регистра **W** будут строго определенными: **0, 1, 2, 3.**
То есть, имеет место быть компактная группа чисел.
Вопрос: "Почему не 1, 2, 3, 4"?
Ответ: потому, что для того чтобы, после исполнения команды **addwf PC,F**, перейти, например, на следующую после нее команду (сценарий № 1), приращение счетчика команд **PC** должно быть нулевым.
Это объясняется тем, что ***счетчик команд PC автоматически инкрементируется при исполнении любой команды*** и поэтому еще раз повторяю, ***если в регистре W, на момент исполнения команды addwf PC,F, записано число N, то после исполнении команды addwf PC,F, произойдет переход рабочей точки программы на команду начала исполнения сценария программы, с номером N+1.***
Например, если в регистре **W** "лежит" число **0**, то произойдет переход на начало исполнения сценария **№1** (команда его выбора находится сразу же после команды **addwf PC,F**). И т.д.
Вывод из этого следующий: к моменту исполнения команды **movf PortA,W**, первые 2 вывода порта А должны быть настроены на работу "на вход", а на выводах **RA2 ... RA4** должны быть

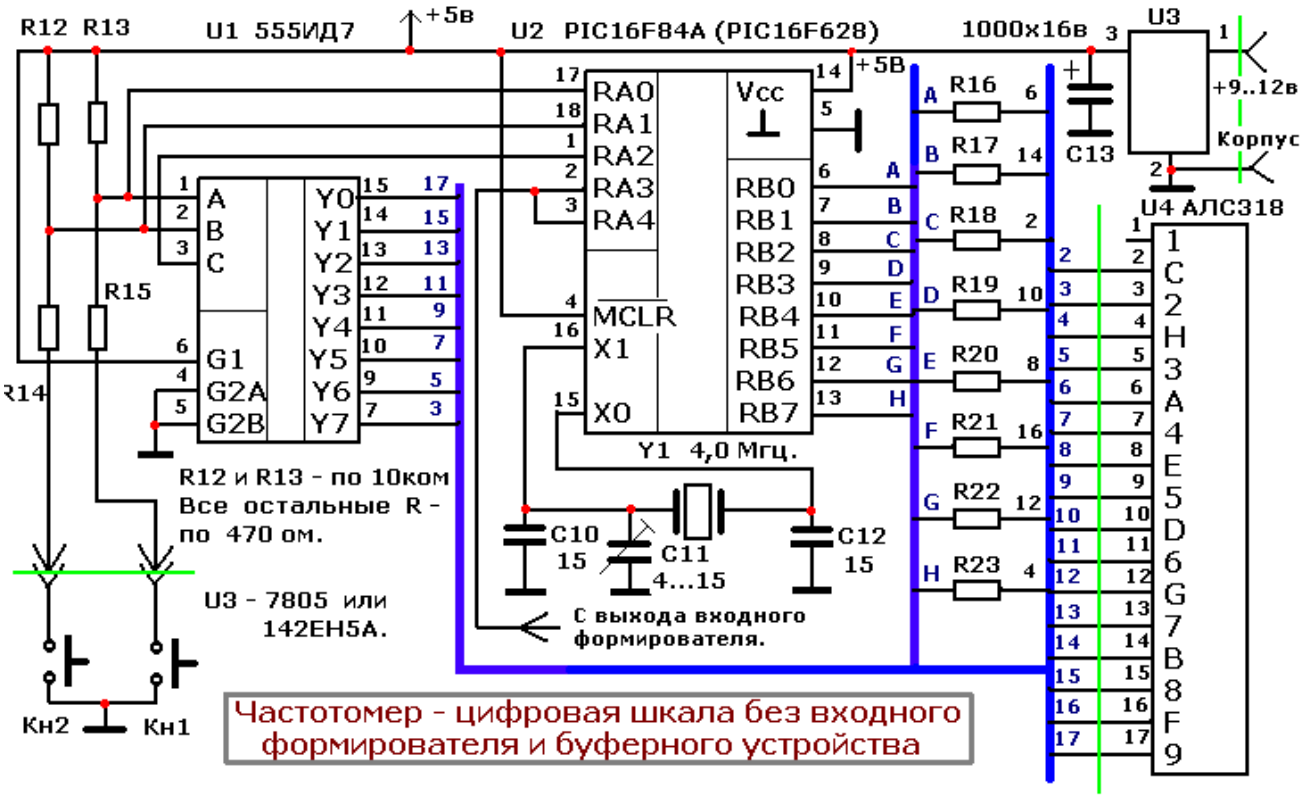
нулевые уровни.
В соответствии с этим, возможные результаты опроса клавиатуры будут следующими:
00000000 - для 1-го сценария (**W**=**0**), обе кнопки нажаты,
00000001 - для 2-го сценария (**W**=**1**), 1-я кнопка нажата, а 2-я отжата,
00000010 - для 3-го сценария (**W**=**2**), 2-я кнопка нажата, а 1-я отжата,
00000011 - для 4-го сценария (**W**=**3**), обе кнопки отжаты.
Примечание: нажатию кнопки соответствует **0**, а отжатию, **1**.
Красным цветом выделено то, что изменяется.
В результате опроса клавиатуры, в регистр **W** записывается одно из этих чисел, и далее происходит приращение содержимого счетчика команд **PC** на величину этого числа, плюс один автоинкремент.
Например, для перехода на **3-й сценарий**, содержимое счетчика команд **PC** увеличивается на **2** (за счет суммирования содержимого счетчика команд **PC** и содержимого регистра **W** = **2**), а затем и еще на **1** (за счет автоинкремента содержимого счетчика команд **PC**, имеющего место быть при исполнении команды **addwf PC,F**).
Пример № 1 (см. **addwfpc.asm**) взят из текста моей программы, которая "обслуживает" частотомер-цифровую шкалу.
Переходы на исполнение первых трех сценариев организованы с помощью команд безусловных переходов (**goto**), а 4-м сценарием является сценарий "программа исполняется далее".
В зависимости от того, какая из первых трех команд **goto** будет исполнена, будет отработана одна из трех подпрограмм (**Function** → 1-й сценарий, **Plus** → 2-й сценарий, **Minus** → 3-й сценарий).
Естественно, что количество сценариев может быть и бОльшим.
В последнем, 4-м сценарии, также можно использовать команду **goto**.
А можно и не использовать (как в приведенном примере).
Это зависит от замысла программы.

В **примере № 2** показана организация так называемого табличного (аналогия с таблицей), вычисляемого перехода, с целью преобразования двоично-десятичного кода в код 7-сегментного индикатора.
При конструировании устройств с отображением данных в одном или нескольких 7-сегментных индикаторах, такого рода преобразование является абсолютно необходимым.
Подпрограмму этого преобразования, построенную по классическому принципу, Вы сейчас и видите (ПП **TABLE**, но можно назвать ее как угодно).
После стандартной команды вычисляемого перехода **addwf PC,F**, компактно расположены 15 команд начала исполнения сценариев программы.
Они короткие.
"Расшифровка" команды **retlw k** : в регистр **W**, записывается константа, после чего, автоматически осуществляется возврат по стеку.
Откуда, что и куда?
Давайте разбираться.
Если речь идет о возврате по стеку, то сначала нужно осуществить условный переход в ПП **TABLE** (**call TABLE**).
В принципе, команда **retlw** это та же команда **return** (возврат из подпрограммы по стеку), но только более "навороченная".
Ее "навороченность" заключается в том, что ***она совмещает в себе рабочую операцию (копирование в регистр W константы) и возврат по стеку.***
При исполнении команды **return**, рабочей операции не совершается, а осуществляется только возврат по стеку.
Таким образом, команду **retlw** можно рассматривать как "микроподпрограмму", состоящую из одной рабочей команды (копирование константы в регистр **W**), после исполнения которой, осуществляется автоматический возврат по стеку.
Вот такая "хитрая" получается команда, и кстати, очень удобная (многое упрощает).
После исполнения текущей команды **retlw** (того или иного сценария), в регистре **W**, "оседает" некое число (то, которое требуется получить в результате реализации текущего сценария, а другими словами – результат кодового преобразования), которое, в дальнейшем, используется в программе.

В рассматриваемом случае, осуществляется кодовое преобразование чисел, "привязанное" к 7-сегментному индикатору, но можно осуществить эту "привязку" и к чему-то другому. Например, к жидкокристаллическим модулям.
В этом случае, тоже требуется кодовое преобразование.
Посмотрите на первые 10 команд **retlw** и обратите внимание на числовые значения констант, которые, для удобства, представлены в бинарном виде.
Сверьтесь с "раскладкой" секторов 7-сегментного индикатора (**A,B,C,D,E,F,H**), и Вы без особого труда заметите, что например, после исполнения команды 1-го сценария, в регистр **W** скопируется число, отображающее в 7-сегментном индикаторе символ **0**, 2-го сценария – символ **1**, 3-го сценария – символ **2** и т.д.
Это означает то, что например, весовой код двоичного числа **0h**, непригодный для отображения в 7-сегментном индикаторе в виде символа **0**, с помощью подпрограммы табличного, вычисляемого перехода, будет преобразован в код, который 7-сегментный индикатор отображает в виде символа **0**.
То же самое относится и ко всем остальным двоичным числам до **9** включительно.
Первые 10 команд **retlw** осуществляют кодовое преобразование двоичных чисел в коды символов **0 … 9**.
Первая команда сверху, производит кодовое преобразование двоичного кода числа **0** в код символа **0**, а последняя, кодовое преобразование двоичного кода числа **9** в код символа **9**.
Принцип работы последующих 5-ти команд → такой же, только осуществляется кодовое преобразование не в символы цифр, а в символы латинских букв (с задействованием запятой), и константы представлены не в бинарной, а в десятичной системе исчисления (можно перевести и в бинарную).
Итак, после исполнения того или иного сценария, в регистре **W** "осело" число, соответствующее коду одного из нескольких символов, которые способен отобразить 7-сегментный индикатор.
Остается только скопировать это число, из регистра **W,** в регистр порта (его выводы должны быть настроены на работу "на выход"), к выводам которого подключены секторы 7-сегментного индикатора.
То есть, вывести символ на индикацию.
Каким образом должны формироваться значения чисел, копируемых в регистр **W**, перед началом отработки подпрограммы табличного, вычисляемого перехода **TABLE**?
Команда **call TABLE** может находиться в тексте программы или до, или после ПП **TABLE**, так как в PIC16F84A (в связи с "слабосильностью", разделения на страницы памяти программ нет), условный (и безусловный тоже) переход можно осуществить из любого "места" текста программы, в любое "места" текста программы.
Единственное ограничение: ***"граница" между блоками памяти программ*** (один блок это 256 ячеек **PC**) ***не должна "проходить" через таблицу вычисляемого перехода*** (об этом говорилось ранее).
В нашем случае, команда **call TABLE** находится выше (по тексту программы, а значит и в **PC**) ПП **TABLE**.
В случае 15-тисценарной работы, с учетом того, что команды начала исполнения сценариев программы расположены компактно, максимальное значение числа, предварительно записываемого в регистр **W**, должно быть 15-1=**14**.
То есть, в регистр **W**, предварительно, должны записываться числа, находящиеся в числовом диапазоне от **.0** до **.14**.
Для первых 10-ти сценариев, значения этих чисел дублируются в символьной форме (**0→0**, **1→1**, **2→2**, ... **9→9**).
В данном случае, к значениям чисел, предварительно записываемым в регистр **W** для исполнения сценариев с **№11** по **15** (числа с **.10** по **.14**), должны быть "жестко привязаны" результаты выполнения операций установки признаков режимов, которые, в свою очередь, "жестко привязаны" к результату опроса состояния клавиатуры (режимы работы устройства переключаются при помощи клавиатуры).
Аналогичные "привязки" режимов/подрежимов работы устройства (или еще чего-то) к тем или иным символам, выводимым на индикацию, осуществляются до исполнения команды **call TABLE** и "механизм" их осуществления может быть различным.
В большинстве случаев, ПП вычисляемого перехода, подобная ПП **TABLE**, вызывается (**call TABLE**) из циклических подпрограмм обработки содержимого неких регистров общего назначения, содержащих в себе данные, предназначенные для вывода на индикацию.

Для краткости, назову эти регистры **LED**ами. От **LED0** (самый младший), до **LED7** (самый старший).
Например, если результат измерения (подсчета) нужно вывести на индикацию как 8-разрядное, десятичное число, то двоичный результат измерения "прогоняется" через двоично-десятичное преобразование (о нем, позднее), в итоге которого, результат измерения помещается в младшие полубайты 8-ми **LED**ов.
Далее, содержимое этих регистров, с соблюдением порядка старшинства, поочередно копируется в регистр **W**, подвергается кодовому преобразованию (**call TABLE**) и выводится на индикацию в виде символов.
Что касается последнего, то "технология" вывода на индикацию проста: результат кодового преобразования, из регистра **W,** копируется в регистр того порта, к выводам которого подключены выводы секторов 7-сегментного индикатора.
А теперь представился удобный случай для того чтобы, в общем виде, разобраться с таким понятием как динамическая индикация.
Для определенности, предположим, что в одном полном цикле индикации, результаты кодового преобразования выводятся на индикацию, начиная с младшего, десятичного разряда линейки 7-сегментных индикаторов (в порядке возрастания старшинства).
По окончании последовательного вывода результатов кодового преобразования во все 8 знакомест линейки, происходит переход (по кольцу) на вывод символа в младшее знакоместо линейки.
И так далее. До конца линейки.
После этого, все опять повторяется снова и снова.
Вот Вам и типичный пример "двойной закольцовки": один "виток" большого кольца индикации равен 8-ми "виткам" малого кольца индикации, "дислоцирующегося" внутри большого кольца индикации.
Если речь идет о фиксированном количестве "витков" (8), то количество "витков" малого кольца индикации нужно "поставить на счетчик".
Количество "витков" большого кольца индикации тоже нужно "поставить на счетчик", ведь нельзя же допустить, чтобы рабочая точка программы все время находилась в подпрограмме вывода данных на индикацию.
В противном случае, "не будут делаться другие, важные дела", например, замер.
А без него, такая индикация и даром не нужна.
То есть, должно быть "отмотано" фиксированное количество "витков" большого кольца индикации, после чего рабочая точка программы должна покинуть ПП вывода данных на индикацию и "сделать другие, важные дела".
После того как "она их сделает", и в результате этого, в наличии будет иметься то, что нужно вывести на индикацию, снова начинается отработка ПП вывода данных на индикацию.
Все повторяется снова. Ну и так далее. Много раз.
В интервале времени каждой такой "отлучки", 7-сегментные индикаторы "гасятся" вплоть до начала следующей отработки ПП вывода данных на индикацию.
Это и является причиной так называемых "мерцаний", являющихся недостатком динамической индикации.
"Заход на второй круг".
В отдельно взятом, 7-сегментном индикаторе, символ индицируется ("высвечивается") в течение интервала времени формирования одного "витка" малого кольца индикации.
В этом интервале времени, он активен.
В начале следующего "витка" малого кольца индикации, этот 7-сегментный индикатор переводится из активного, в пассивное состояние ("гасится"), а следующий, по разрядности, 7-сегментный индикатор активируется.
Ну и так далее. До окончания активации последнего, 7-сегментного индикатора линейки.
Таким образом, речь идет о последовательном, поразрядном выводе результатов измерения (счета) на индикацию.
"Привязка" конкретных, десятичных разрядов результата измерения (счета) к конкретным знакоместам линейки 7-сегментных индикаторов, осуществляется путем последовательной (в порядке старшинства) "запитки" 7-сегментных индикаторов, входящих в состав линейки.
В зависимости от того, какие именно 7-сегментные индикаторы применены (с общим катодом или с общим анодом), эта "запитка" осуществляется либо коммутацией точки соединения общих выводов сегментов на плюс источника питания, либо ее коммутацией на корпус.

Остальные выводы сегментов объединяются в группы (по принципу одноименности секторов) и в пределах этих групп, "параллелятся".
Получается 8 выводов, которые и подключаются к выводам порта, который задействован для управления линейкой.
Сказанное относится к тому случаю, если линейка создается из отдельных, 7-сегментных индикаторов.
Если речь идет о промышленных вариантах такой линейки, то ничего "параллелить" не нужно, так как разработчики об этом позаботились.
Для того чтобы обеспечить нужный порядок коммутации 7-сегментных индикаторов линейки, необходим дешифратор, то есть, устройство, у которого может быть активен только один из его выходов.
То, какой именно из них будет активен, зависит от адресного кода, подаваемого на его входы управления.
При наличии необходимого количества выводов портов, такой дешифратор можно реализовать программно (позднее, будет показано, как это делается).
Но не всегда выводов портов хватает.
Например, **PIC16F84A**.
Все 8 выводов порта B задействованы (через них осуществляется вывод символов на индикацию).
Остается 5 выводов порта A.
Но для линейки из 8-ми 7-сегментных индикаторов, этого явно недостаточно, так как нужно 8.
Да еще и под другие нужды нужно что-то выделить.
Поэтому ничего другого не остается, как применить внешний (по отношению к ПИКу) дешифратор на 3 адресных входа и 8 выходов.
Это уже вполне приемлемо, так как 3 это не 8. "Извернуться" можно.
Составив соответствующим образом программу, на этих 3-х входах, можно программно сформировать сигналы управления дешифратором, который, в свою очередь, будет осуществлять необходимые коммутации.
Ниже Вы видите практическую иллюстрацию сказанного.



Это принципиальная схема частотомера-цифровой шкалы.
На ней Вы видите то, о чем говорилось выше: выводы порта B подключены к выводам секторов 7-сегментных индикаторов (одноименные секторы "запараллелены" внутри АЛС318), 8 общих выводов 7-сегментных индикаторов подключены к выходам дешифратора

(555ИД7), а адресные входы дешифратора подключены к первым трем выводам порта А.
Резисторы **R16...R23** - гасящие.
Они нужны для предотвращения токовых перегрузок оконечных каскадов защелок порта В.
Пояснение.
В примере №1, говорилось о реализации вычисляемого перехода, осуществляемого по внешнему, управляющему сигналу, сформированному клавиатурой, и имелось ввиду то устройство, принципиальную схему которого Вы видите выше.
Посмотрите на схему.
Кнопки клавиатуры подключены к выводам **RA0** и **RA1**.
Значит, эти выводы должны быть настроены на работу **"на вход"**.
Но к этим же выводам подключены и первые два адресных входа дешифратора 555ИД7.
Значит, эти выводы должны быть настроены на работу **"на выход"**.
Вопрос: "Как же тогда все это "мирно сосуществует" и работает в комплексе"?
Ответ: бОльшую часть полного цикла программы, выводы **RA0** и **RA1** работают **"на выход"**, то есть, управляют дешифратором (обеспечивают процесс динамической индикации).
Для опроса клавиатуры, требуется очень малое время (доли процента от времени полного цикла программы).
В данном случае, опрос клавиатуры производится в начале полного цикла программы и непосредственно перед этим опросом, выводы **RA0** и **RA1** программно "перенастраиваются" с работы **"на выход"** на работу **на "вход"**, а после окончания опроса сразу же программно "перенастраиваются" с работы **"на вход"** на работу **"на выход"**.
Эти "перенастройки" происходят очень быстро.
Вот Вам и типичный пример того, как одни и те же выводы портов задействуются для выполнения функционально различных действий, а "рулит всем этим делом" программа.
По этому же принципу, можно задействовать группу выводов портов и для выполнения большего количества различных "функциональностей" (возможность этого нужно рассматривать в каждом конкретном случае), что, согласитесь со мной, весьма существенно.
И особенно для "слабосильных" микроконтроллеров, имеющих небольшое количество выводов портов.
В данном же случае, без такого рода комплексного "задействования" просто не обойтись, так как если выделять для каждой "функциональности" отдельные группы выводов порта А, то пяти его выводов окажется недостаточным и придется задействовать микроконтроллер с бОльшим количеством выводов портов.

Работа с EEPROM памятью данных.

EEPROM память данных так же, как и память программ, является энергонезависимой памятью, но к памяти программ она отношения не имеет, и по этой причине, она как бы "стоит особняком" (выполняет свои специфические функции).
Ее функции, в конечном итоге, сводятся к сохранению либо констант, либо числовых результатов операций после выключения питания микроконтроллера, с целью их дальнейшего использования после следующего включения питания.
Например, в моем частотомере-цифровой шкале, EEPROM память данных задействована в двух случаях: в нее записывается, определяемое пользователем (и по умолчанию), значение промежуточной частоты, которое в дальнейшем используется при формировании показаний индикатора в режиме цифровой шкалы, и она же выполняет функции энергонезависимой памяти настроек (установка, после включения питания, тех режимов работы прибора, которые были установлены на момент предшествующего выключения питания).
В демонстрационной версии программы, которая предлагалась до замены ее на рабочую, EEPROM память данных дополнительно задействовалась и в системе блокировки, которая срабатывала на третьем, после прошивки, включении питания, и в системе защиты от несанкционированных попыток изменения содержимого счетчика разрешенного количества включений питания.
Это только отдельный пример.
Есть еще масса всевозможных вариантов задействования EEPROM памяти данных, в результате которых, любое устройство на микроконтроллере можно сделать более "интеллектуальным" и удобным в использовании или/и ввести некие, трудно устранимые пользователем, ограничения, проявляющиеся после "выхода за границы дозволенного".
Что касается ограничений, то это одно из моих "хобби".

И не по той причине, что я "враг народа". Просто интересно.
В связи с этим, спешу успокоить тех людей, которые пользуются моими "прошивками": никаких "сюрпризов" они не содержат и на этот счет можно быть абсолютно спокойным.
EEPROM память, сама по себе, "тупая как валенок".
Из нее можно только или считать данные, или записать их в нее.
Если, при ее задействовании, речь идет о какой-то "интеллектуальности", то эту "интеллектуальность" нужно "закладывать" в рабочую часть программы.
В простейшем случае, данные (числа) в EEPROM память записываются при "прошивке" ПИКа, а в ходе выполнения программы они периодически считываются из нее.
При выключении питания, эти данные не уничтожаются и с ними можно работать при последующих включениях питания.
В частотомере-цифровой шкале, этот вариант задействования EEPROM памяти применен для установки первичного значения промежуточной частоты (по умолчанию).
Другой, более востребованный случай: нужные программисту данные, могут быть записаны в EEPROM память данных, а также считаны из нее, при помощи специальных подпрограмм, рекомендованных разработчиками.
Запись и/или чтение данных можно "спровоцировать" различными способами.
Таким образом, можно многократно, по желанию пользователя или автоматически, записывать в EEPROM память данных различные числа, которые, после считывания их из нее, используются в работе программы и которые сохраняются при выключении питания.
В частотомере-цифровой шкале, этот вариант задействования EEPROM памяти применен для организации режима работы типа "запись в EEPROM память значения промежуточной частоты, определяемой пользователем".
Данные, в EEPROM память данных, могут быть записаны при "прошивке" ПИКа, а затем, по ходу исполнения программы, считаны.
После этого, они могут быть видоизменены, и в этом виде, снова записаны в EEPROM память данных.
Таких циклов "считывания/записи" может быть много и они будут происходить до тех пор, пока результат чтения, например, не "войдет в зону" чисел, которую задал программист.
По факту этого события, производится какое-то действие.
С привлечением EEPROM памяти данных, можно организовать парольный доступ к чему-либо.
Можно еще перечислить множество различных случаев задействования EEPROM памяти данных.
Во всех случаях, "рулить" ей будет либо рабочая часть программы, либо она же, плюс директивы записи в EEPROM память данных, располагающиеся в "шапке" программы.
Что это за директивы, Вы узнаете ниже.
Естественно, что сейчас я не буду "обрушивать на Вас лавину информации", связанной с рассмотрением сложных случаев задействования EEPROM памяти данных.
Примеры я привел только с той целью, чтобы показать Вам перспективность работы с EEPROM памятью данных.
Для начала, нужно понять, "что это вообще такое и с какого бока к этому подойти".
Объем EEPROM памяти данных не велик.
Для **PIC16F84A**, это **64 ячейки**. В других типах ПИКов, может быть и больше.
В каждую из этих ячеек может быть записан один байт (число от **.00** до **.255**).
После записи, этот байт (байты) можно считать и результат этого считывания использовать в программе.
Каждая из этих 64-х ячеек, в диапазоне адресов от **.00** до **.63** (**00h .. 3Fh**), имеет ***свой адрес***, ***который обязательно нужно указывать как при чтении из EEPROM памяти данных, так и при записи в нее.***
По ходу составления программы, программист определяет, содержимое какой именно ячейки (ячеек) нужно считать, и в какую именно ячейку (ячейки) нужно произвести запись, с "привязкой" этих действий к логике программы.
Запись, в ту или иную ячейку EEPROM памяти данных, того или иного числа производится ***"по верху"*** "старого" числа.
При этом, "старое число погибает".
То есть, речь идет о ***замене одного числа другим.***
Во время "прошивки" ПИКа, в EEPROM память данных может быть произведена предварительная запись данных.

Этим "рулит" директива (или несколько таких директив) **DE** (см. ниже), которая, в "шапке" программы, должна располагаться до директивы **org 0**.
Поставим перед собой достаточно простую и конкретную задачу (что-то типа задания на разработку).
До начала исполнения рабочей части программы, необходимо записать, в ячейку, например, с адресом **02h**, число, например, **.100** (**64h**).
Затем, в ходе исполнения программы, нужно скопировать содержимое этой ячейки (**.100**) в регистр регистр общего назначения с названием, например, **Registr**, после чего, произвести какую-нибудь операцию с содержимым этого регистра.
Например, увеличим его содержимое на единицу (**.101**), с сохранением результата операции в этом же регистре **Registr**.
Затем, запишем число **.101** в ту же ячейку EEPROM памяти данных.
По ходу этих "разборок", я буду объяснять общие положения работы с EEPROM памятью данных.
Обращаю Ваше внимание на следующую, хотя и достаточно примитивную, но "имеющую право на жизнь", аналогию, которая поможет Вам понять суть работы с EEPROM памятью: ячейки EEPROM памяти можно представить себе в виде специфических регистров общего назначения.
И в самом деле, с содержимым этих ячеек можно производить те же операции, что и с содержимым регистров общего назначения (имеется ввиду чтение или запись, и не более того), и если не требуется сохранить данные после выключения питания, то задачу, сформулированную выше (а также и множество других задач), можно решить, задействуя только регистры общего назначения, что, в этом случае, всегда и делается на практике (оперативная память).
К "услугам" EEPROM памяти данных прибегают только тогда, когда "деваться некуда" (если не сохранить данные после выключения питания, то не будет реализована задумка программиста).
Главное отличие ячеек EEPROM памяти данных, от регистров общего назначения, состоит в том, что в них сохраняются данные после выключения питания.
Это хорошо, но не "шустро".
В том смысле, что процедуры чтения/записи (особенно записи) довольно-таки "громоздки" (достаточно большое количество команд), и они отрабатываются не за 1 м.ц.
И даже не за 5, а поболее. Это и есть специфика.

Файл с примером работы с EEPROM памятью данных eeprom.asm находится в папке ***"Тексты программ".***

Это выглядит так:

```
;*******************************************************************************
;                          РАБОТА С EEPROM ПАМЯТЬЮ ДАННЫХ
;*******************************************************************************
;                                    "ШАПКА ПРОГРАММЫ"
;===============================================================================
;.....................................................
;.....................................................
;===============================================================================
; Определение положения регистров специального назначения.
;===============================================================================
Intcon      equ         0Bh         ; Регистр Intcon.
Status      equ         03h         ; Регистр Status.
EEData      equ         08h         ; EEPROM - данные
EECon1      equ         08h         ; EECON1 - банк1.
EEAdr       equ         09h         ; EEPROM - адрес
EECon2      equ         09h         ; EECON2 - банк1.
;.....................................................
;.....................................................
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
Registr     equ         0Ch         ; Регистр оперативной памяти, используемый
```

```
                                ; при работе с EEPROM.
;.....................................................
;.....................................................
;=======================================================================
; Определение места размещения результатов операций.
;=======================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;=======================================================================
; Присваивание битам названий.
;=======================================================================
RP0         equ         5           ; Бит выбора банка.
GIE         equ         7           ; Бит глобального разрешения прерываний.
;=======================================================================
            org         2100h       ; Обращение к EEPROM памяти данных.
            DE          0h,0h,64h   ; Записать в ячейки с адресами .0, .1, .2
                                    ; числа 0h, 0h, 64h (.100) соответственно.
            DE          0h,0h,0h    ; Записать в ячейки с адресами .3, .4, .5
                                    ; числа 0h, 0h, 0h соответственно.
            DE          "Korabelnikow E.A. Rus. Lipetsk. 03.2005"  ; Записать
                                    ; символы в ячейки с адресами с .6 по .46
;=======================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;***********************************************************************

-------------------------------------------------------------------------
                        ПОЯСНЕНИЯ ПО ДИРЕКТИВЕ DE
-------------------------------------------------------------------------
                                              Содержимое EEPROM
1. Для  случая, указанного ---> 00 00 64 00 00 00 xx xx   xx xx xx xx xx xx xx xx
   в "шапке" программы     ---> xx xx xx xx xx xx xx xx   xx xx xx xx xx xx xx xx
                           ---> xx xx xx xx xx xx xx xx   xx xx xx xx xx xx xx FF
                           ---> FF FF FF FF FF FF FF FF   FF FF FF FF FF FF FF FF
   xx - 16-ричные числа символов, включая и пробелы.
-------------------------------------------------------------------------
2. А можно и так:
            org    2100h   ---> 00 00 64 FF FF FF FF FF   FF FF FF FF FF FF FF FF
            DE     0h,0h,64h--> FF FF FF FF FF FF FF FF   FF FF FF FF FF FF FF FF
                           ---> FF FF FF FF FF FF FF FF   FF FF FF FF FF FF FF FF
                           ---> FF FF FF FF FF FF FF FF   FF FF FF FF FF FF FF FF
-------------------------------------------------------------------------

;***********************************************************************
;                           РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;***********************************************************************
;.....................................................
;.....................................................
;=======================================================================
;                           Чтение данных из EEPROM
;=======================================================================
; Считывание содержимого байта с адресом 02h и запись его в регистр общего
; назначения Registr.
;-----------------------------------------------------------------------
            bcf         Status,RP0  ; Переход в нулевой банк.

            movlw       2           ; Записать в регистр W константу 02h.
            movwf       EEAdr       ; Скопировать 02h, из регистра W,
                                    ; в регистр EEAdr.

            bsf         Status,RP0  ; Переход в первый банк.
            bsf         EECon1,0    ; Инициализировать чтение.
            bcf         Status,RP0  ; Переход в нулевой банк.

            movf        EEData,W    ; Скопировать число из ячейки EEPROM,
                                    ; с адресом 02h, в регистр W.
```

```
            movwf       Registr     ; Скопировать число из регистра W
                                    ;в регистр Registr.
;-------------------------------------------------------------------------
; Изменение содержимого регистра Registr.
;-------------------------------------------------------------------------
            incf        Registr,F   ; Увеличить на 1 содержимое регистра Registr с
                                    ; сохранением результата в нем же.
;=========================================================================
;                           Запись данных в EEPROM.
;=========================================================================
; Запись содержимого регистра Registr в ячейку EEPROM с адресом 02h.
;-------------------------------------------------------------------------
            bcf         Intcon,GIE  ; Глобальный запрет прерываний.

            movlw       2           ; Записать, в регистр W, константу 02h.
            movwf       EEAdr       ; Скопировать константу 02h, из регистра W,
                                    ; в регистр EEAdr.
            movf        Registr,W   ; Скопировать число, из регистра Registr,
                                    ; в регистр W.
            movwf       EEData      ; Скопировать число, из регистра W,
                                    ; в ячейку EEPROM с адресом 02h.
            bsf         Status,RP0  ; Переход в первый банк.
            bsf         EECon1,2    ; Разрешить запись.

            movlw       055h        ; Обязательная
            movwf       EECon2      ; процедура
            movlw       0AAh        ; при записи.
            movwf       EECon2      ; ----"----
            bsf         EECon1,1    ; ----"----

            bcf         EECon1,4    ; Сбросить флаг прерывания по окончании
                                    ; записи в EEPROM.
            bcf         Status,RP0  ; Переход в нулевой банк.
;-------------------------------------------------------------------------
;.........................................................................
;.........................................................................
;*************************************************************************
            end                     ; Конец программы.

Примечание: регистр EECon2 не реализован физически. Он используется в операциях
            записи в EEPROM память данных, для реализации обязательной
            последовательности команд.
```

Этот файл не является "полноценной" программой.
Проекта под него создавать не нужно, а просто откройте его (***File → Open***).
Вы видите "заготовку" программы, с задействованием EEPROM памяти данных, в которой решается задача, сформулированная выше.
Все что относится к работе с EEPROM памятью данных, расписано подробно.
Остальная часть программы обозначена условно → строками из точек.
Это относится и к "шапке" программы, и к ее рабочей части.
На месте этих строк, образно выражаясь, может быть "все что угодно".
Это "все что угодно", в данный момент, не существенно.
Нужно разобраться с тем, как предварительно записать, в ячейку EEPROM памяти данных с адресом **02h**, число **.100**, затем, в рабочей части "программы", скопировать это число в регистр оперативной памяти, увеличить его на **1**, и после этого, записать число **.101** в ту же ячейку EEPROM памяти данных.
Сначала разберемся с предварительной записью.
Предварительная запись данных, в ячейки EEPROM памяти данных, производится при помощи директивы DE.
Прежде чем применить эту директиву, необходимо обратиться к EEPROM памяти данных (активизировать, включить, запустить → кому как нравится) при помощи директивы **org** с указанием адреса **2100h** (см. текст примера).
Директива **org 2100h** всегда трактуется однозначно (других вариантов "включения"

предварительной записи, в EEPROM память данных, нет), и поэтому можно "не забивать себе голову" информацией о том, что такое **2100h**, а просто принять это как данность (запомнить как правило).
После этой директивы, можно исполнять одну или несколько директив **DE**.
При помощи директивы **DE**, в ячейки EEPROM памяти данных, записываются числа в диапазоне от **.00** до **.255**.
Вопрос: "По каким адресам и в каком порядке"?
Ответ: первая, после директивы **org 2100h**, директива **DE**, записывает первое, указанное в рабочей части этой директивы, число (крайнее левое), в ячейку EEPROM памяти данных с адресом **.00** (**00h**), второе число записывается в ячейку с адресом **.01** (**01h**), третье, в ячейку с адресом **.02** (**02h**) и т.д., вплоть до последней ячейки (если в этом есть необходимость).
Значения чисел, предварительно записываемых в ячейки EEPROM памяти данных, должны быть указаны в рабочей части директивы DE и расположены в порядке следования адресов, по которым эти числа нужно записать (начиная с нулевого и далее по порядку), с отделением этих чисел, друг от друга, запятыми.
В данном случае, первая директива **DE** записывает числа **0h, 0h, 64h** (**.100**) в 1, 2, и 3 ячейки соответственно (по адресам **00h**, **01h**, **02h**).
Примечание: обратите внимание на форму представления числа ноль (**0h**).
00h, 000h, 0000h и т. д., это то же самое, что и **0h**. Все незначащие нули, находящиеся слева от значащей цифры, можно не указывать, но крайняя, правая значащая цифра (цифры) указываться должна.
Например, **.0034** = **.034** = **.34** = **0022h** = **022h** = **22h.**
Для того чтобы, в конечном итоге, вся эта "канцелярия" не вызывала у Вас затруднений, по ходу дальнейшей работы, я буду "манипулировать" различными формами представления чисел.
В нашем случае, число **.100** (**64h**) нужно записать в 3-ю ячейку с адресом **02h**.
Так как первая, после директивы **org 2100h**, директива **DE** "жестко привязана" к "нулевой точке отсчета" (адрес ячейки EEPROM памяти данных, равный **0**), то для того чтобы "дойти" до третьей ячейки, нужно чем-то заполнить первые две ячейки (записать в них какие-то числа).
Если бы эти 2 ячейки в программе задействовались, то в них нужно было бы записать определяемые программистом числа, но в нашем случае, они не задействованы, и то, какими именно числами они будут заполнены, не имеет значения.
В них можно записать любые, "дозволенные" числа, но обычно, в такого рода "пустышки", записываются нули, что Вы и видите в тексте "программы".
По умолчанию, во все ячейки EEPROM памяти данных, записываются числа **FFh** (кроме ячеек, заполняемых при помощи директивы **DE**), и на этом "фоне", тот "сектор" EEPROM памяти, в котором записаны нули, "сразу бросается в глаза", что делает более комфортным визуальное восприятие содержимого области EEPROM памяти данных.
Для того чтобы показать, "как плодятся и размножаются пустышки", в данном случае, я специально поступил нерационально.
Это я еще "ушел" не далеко (всего-лишь адрес **02h**), а если задействовать, например, последнюю ячейку с адресом **3Fh**?
В этом случае, рабочая часть директивы **DE** будет представлять собой длиннющую строку, заполнение которой, по своей сути, является достаточно бестолковой работой.
Практический вывод из этого следующий: если необходимо задействовать ячейки EEPROM памяти данных, то с точки зрения рациональности, их нужно задействовать, начиная с первой ячейки (адрес **0**) и далее, по-порядку (если нужны несколько ячеек).
В этом случае, до задействованной ячейки не нужно "добираться, плодя пустышки" и рабочая часть директивы **DE** будет компактной и удобной для восприятия.
Если руководствоваться этими соображениями, то в нашем случае, нужно задействовать не 3-ю, а 1-ю ячейку EEPROM памяти данных (в рабочей части программы обращаться не к ячейке с адресом **2**, а к ячейке с адресом **0**).
В этом случае, рабочая часть директивы **DE** будет содержать в себе не 3 числа, а одно число **64h** (**.100**), и дело с концом.
Далее, можно переходить к "разборкам" с рабочей частью программы, а директива **DE** будет первой и последней.
Конечно же, предлагаемый Вашему вниманию пример программы, можно составить и таким образом (на практике, так и нужно делать), но в обучающих целях, пусть она будет такой,

какой есть, а иначе отсутствует предмет разбирательств.
Вопрос: неужели "пустышки" так уж и бесполезны?
Ответ: нет.
Представьте себе такую ситуацию: на момент начала составления текста программы, программист точно знает, что "железобетонно" потребуется одна ячейка EEPROM памяти данных, в которую, в процессе "прошивки", нужно записать константу **64h**, но по ходу составления текста рабочей части программы, возникает необходимость в задействовании еще нескольких групп ячеек, например, двух групп, состоящих из двух ячеек каждая.
В этом случае, директиву **DE** можно "оформить" так: **DE 0h,0h,64h,0h,0h,0h** или так: **DE 0h,0h,0h,0h,64h,0h**, или так: **DE 0h,0h,64h,01h,01h,02h**, или еще как-то (по своему желанию), с целью того чтобы как бы "зарезервировать" ячейки под будущие операции с EEPROM памятью данных, а заодно и визуально выделить, из общей "массы" (на "фоне" **FFh**), группу ячеек, с которыми, в дальнейшем, будет работать программа (элемент удобства, и не более того).
Например, **DE 0h,0h,64h,01h,01h,02h** можно истолковать так: ячейки **1**, **2** и ячейки **4, 5** зарезервированы под "еще не рожденные" операции с использованием EEPROM памяти данных, в 3-ю ячейку, при "прошивке", "закладывается" заранее определенная константа, с числовым значением **.100**, а 6-я ячейка, это помеченный цифрой 2 (а можно и другой), "ефрейторский зазор" (на всякий случай).
Короче, подобного рода "оформление" - дело вкуса.
Я говорю об этом по той причине, что подобного рода "архитектурные излишества" часто запутывают. Особенно начинающих.
Только ради одного этого, такие "разборки" нужны.
Замечания по оформлению.
Например, **DE 0h,0h,64h,0h,0h,0h**.
Если поместить "все это добро" на свое "штатное место", то своей правой частью она "наедет" на "виртуальную", вертикальную линию, на которой находятся точки с запятыми (после них начинаются комментарии).
По этой причине, текст соответствующего комментария придется сдвинуть вправо, что в тексте программы, создает "легкий бардачок".
Конечно же, это абсолютно не "смертельно" (MPLABу это "по барабану") и в работе программы ничего не изменится, но "в оформительском смысле", получается "кто в лес, кто по дрова".
Для людей, приученных к порядку, это примерно то же самое, что сесть на кнопку.
Чтобы "навести в этом бардаке порядок", одну директиву **DE 0h,0h,64h,0h,0h,0h** можно разделить на 2 части так, как это сделано в примере программы.
Если рабочая часть директивы **DE** содержит большее количество чисел, то по такому же принципу, одну эту директиву можно "расчленить" на 3, 4, 5 и т.д. частей, использовав 3, 4, 5 и т.д. директив **DE**.
В этом случае, "начальной точкой отсчета" адреса нижней директивы, будет последний адрес директивы, располагающейся "одним этажом выше", плюс единица.
Директива **DE** может работать не только с числами, указанными в явном виде, но и с символами.
Это означает то, что если в рабочей части директивы **DE** указать символы, входящие в состав стандартного, "компьютерного" кода, то в ячейки EEPROM памяти данных будут записаны числовые значения символов.
MPLAB об этом "позаботится" ("в автомате").
При этом, группу символов нужно "закавычить".
Посмотрите в текст программы.
В нем Вы увидите, что при помощи третьей сверху директивы **DE**, я вставил в незадействованные ячейки (начиная с 7-й) комментарий в символьном виде.
Такого рода "архитектурные излишества" реализуются по принципу типа "если есть свободное местечко, то дай-ка я в него что-нибудь засуну".
Естественно, что это совсем не обязательно.
Если нужно разбить текст такой надписи на части (если она слишком длинная), то это делается по принципу, описанному выше, только каждая группа символов заключается в кавычки.
В тексте программы, я этого делать не стал.
Вывод: использующиеся в программе ячейки, целесообразно расположить в начале

EEPROM памяти данных (начиная с нулевого адреса), а в незадействованных ячейках, по умолчанию, записываются числа **FFh**.
Примечание: атрибут 16-ричной системы исчисления (**h**) не указывается.
Другое дело, если нужно "спрятать/замаскировать", в EEPROM памяти данных, некую "секретную" ячейку (ячейки).
В этом случае как раз и потребуется то, о чем говорилось выше.
Такого рода "секретные" ячейки, в области EEPROM памяти данных, можно "разбросать" (если их несколько) как угодно, "заполнив промежутки" между ними какой-нибудь, сбивающей с толка, "абракадаброй".
Усвоив то, о чем шла речь выше, Вы без особых проблем разберетесь с тем, как это делается.
Проконтролировать содержимое области EEPROM памяти данных можно так.
В главном меню **MPLAB**, нужно щёлкнуть по слову ***Window***, а в раскрывшемся списке, щёлкнуть по строке ***EEPROM Memory***.
Откроется окно ***EEPROM Window***, в котором Вы увидите область EEPROM памяти данных.
В ней можно определить и адрес, и содержимое любой ячейки.
Если после открытия файла примера "программы", Вы откроете окно ***EEPROM Window***, то во всех ячейках, Вы увидите значения чисел, установленных по умолчанию **FFh**, хотя, казалось бы, в соответствии с текстом программы, в части из них, должны "лежать" числа, отличные от **FFh**.
В чем дело?
А дело в том, что если ассемблирования текста программы ни разу не производилось, то в окне ***EEPROM Window*** Вы увидите, во всех ячейках, значения чисел, установленных по умолчанию (**FFh**).
Если текст программы проассемблирован, то в том случае, если EEPROM память данных в программе не задействована, то будет то же самое.
Если проассемблирован текст программы, в которой ячейки EEPROM памяти данных задействованы, то после ассемблирования, Вы увидите, в этих ячейках, те числовые значения, которые "дислоцируются" в рабочей части директивы **DE** (в 16-ричной форме исчисления, без указания буквы **h**).
Если это символы, то они будут указаны в виде чисел.
Обращаю Ваше внимание на следующий, **общий принцип: *в окнах, показывающих содержимое области оперативной памяти, памяти программ и EEPROM памяти данных, отражается результат последнего ассемблирования.***
То есть, если в текст программы внесены изменения, и после этого, ассемблирование не производилось, то проконтролировать эти изменения нельзя.
Вопрос: "Если изменения вносятся не в текст программы, а в комментарии или в оформление программы (во все то, что располагается правее точек с запятыми), то нужно ли, для того чтобы сохранить эти изменения, производить ассемблирование"?
Ответ: есть 2 варианта сохранения изменений в комментариях и/или в оформлении программы.
Первый вариант: произвести ассемблирование.
В этом случае, в **HEX-файле** ничего не изменится (**MPLAB** "не видит" ничего, что расположено правее точек с запятыми).
Значимая часть программы, в которую не внесены изменения, просто "перезапишется", но при этом, будет обновлено то, что расположено правее точек с запятыми.
Второй вариант: так как **HEX-файл** обновлять не нужно, то можно сохранить, указанные выше изменения, только в **ASM-файле**.
Для этого, нужно произвести стандартное действие: в главном меню **MPLAB**, нужно щёлкнуть по слову ***File***, а в выпадающем списке, по строке ***Save***.
На первых порах, проще всего (чтобы не "распылять внимание"), после любых изменений, внесенных в тексте программы, производить ассемблирование.
Вернемся к окну ***EEPROM Window***.
Для того чтобы увидеть результат использования директивы **DE**, с учетом сказанного, текст программы (файл **eeprom.asm**) нужно проассемблировать.
Если Вы попробуете сделать это, то получите сообщение о 21 ошибке.
При этом, в окне ***EEPROM Window*** ничего не изменится (во всех ячейках **FFh**), что является закономерным результатом неисполнения требований, предъявляемых к оформлению текста программы.

MPLAB "заругался" абсолютно обоснованно, так как программа неправильно оформлена и поэтому, "с его точки зрения" (и по факту тоже), она является "неполноценной".
Вопрос: а как же в этом случае, при помощи симулятора, убедиться в том, что директивы **DE** исполняются?
Ответ: никак.
Вывод: ***проверить результаты исполнения директив DE можно только в том случае, если текст оформлен в виде "полноценной" программы, и этот текст проассемблирован без ошибок.***
Это относится не ко всем директивам.
Например, результаты исполнения директив **equ**, **end** (и т.д.), в симуляторе, проконтролировать нельзя, так как контролируются изменения только того, что отображается в "контрольных" окнах, а не результаты действий, обеспечивающих функционирование программы.
Но это не свидетельствует о том, что **MPLAB** "самоустранился от этого дела". Это не так.
Если "что-то пошло наперекосяк", то **MPLAB**, после ассемблирования, выдаст соответствующие сообщения об ошибках и/или предупреждения.
Обращаю Ваше внимание на пояснения, находящиеся в тексте примера между "шапкой" "программы" и ее рабочей частью.
Вы видите варианты заполнения ячеек EEPROM памяти данных, для 2-х случаев "оформления" директив **DE**.
Это именно то, что будет наблюдаться, в окне ***EEPROM Window***, после проведения успешного ассемблирования, если предположить, что текст примера оформлен как "полноценная" программа.

Разбираем текст программы.

До и после интересующих нас операций, программа может работать множеством различных способов и реализовывать какие угодно замыслы программиста.
В данном случае, это не важно.
Важно то, как именно происходит работа с EEPROM памятью данных.
На момент начала исполнения рабочей части программы, 46 ячеек EEPROM памяти заполнены числами ("сработала" директива **DE**), а остальные ячейки заполнены числами, устанавливаемыми по умолчанию (**FFh**).
Из всех этих ячеек, нас интересует только одна - третья, с адресом **02h**, так как в ней, предварительно, записано число, с которым нужно производить интересующие нас операции (**64h** или **.100**).
В этом смысле, все остальные, предварительно записанные числа, можно рассматривать как "архитектурные излишества", не влияющие на ход исполнения программы.
В данном случае, уходов в прерывания нет.
Исполнение программы начинается стандартно. С ПП **START**.
В рабочей части программы, я не стал обозначать начало этой ПП. Имейте это ввиду.
После "запуска в эксплуатацию" рабочей части программы, она исполняется до тех пор, пока не возникнет необходимость в работе с EEPROM памятью данных.
Если такая необходимость имеется, то рабочая точка программы должна "перейти" на начало исполнения группы команд, которая работает с EEPROM памятью данных.
В данном случае, переход на это "начало" происходит "естественным образом" (на "линейном участке программы". Без задействования **call**/**goto**).
Если используются команды переходов **call**/**goto**, то переход на это "начало" может происходить из любого "места" программы.
В этом случае, группе команд, обеспечивающей работу с EEPROM памятью данных, нужно либо "присвоить статус" подпрограммы, либо выставить метку на первой команде этой группы.
Разбираемся с тем, что же должна из себя представлять эта группа команд?
В соответствии со сформулированной выше задачей, это должна быть группа команд, обеспечивающая считывание числа, из 3-й ячейки EEPROM памяти данных, в регистр оперативной памяти.
Следовательно, под это дело, нужно задействовать один из регистров общего назначения.
Назначаем ему название, например, **Registr** и "прописываем" его в "шапке" программы по адресу, например, **0Ch**.

Теперь мы конкретно знаем, откуда считывать и куда "закладывать на временное хранение" результат этого считывания.
Остается только выяснить, как конкретно это сделать?
Вы видите перед собой "кучу туманных" команд и по всей видимости, думаете, что "разборки" с ними будут достаточно трудоемкими.
Ничего подобного.
Группа команд чтения данных из EEPROM памяти данных (и записи тоже) является **стандартным "набором" команд, рекомендованных к применению разработчиками**.
То есть, "изобретать колесо" не нужно.
Нужно только "врезать" эту стандартную группу команд в нужное "место" текста программы.
Внутри этой группы команд, нужно:
- ***выставить адрес ячейки, из которой производится чтение (в нашем случае, 02h),***
- ***и указать название регистра, в который запишется считываемое число (в нашем случае, Registr).***

О том, что после этого будет происходить в "дебрях" ПИКа, программисту знать не обязательно.
Если группа команд чтения не содержат ошибок, то и беспокоиться не о чем.
То же самое относится и к процессу записи данных в EEPROM память данных.
Тем не менее, хотя бы "для общего развития", разобраться с этим стандартным набором команд стОит.
Чтение данных, из EEPROM памяти данных, начинается с перехода в 0-й банк (**bcf Status,RP0**).
Если в ходе составления программы выяснится, что на момент начала процедуры чтения, происходит работа в 0-м банке, то команду **bcf Status,RP0** из текста программы можно (и нужно) удалить (какой смысл выбирать 0-й банк, если работа и так происходит в нем?).
А можно и оставить все как есть (это для ленивых).
Следующие 2 команды - стандартные команды записи константы, о которых многократно говорилось ранее.
В данном случае, константой является адрес 3-й ячейки (**02h**).
Обратите внимание на то, что число **2** указано без атрибута системы исчисления.
В этом случае, оно "воспринимается" как **02h** (о том, в каких случаях можно указывать число без атрибутов, я объяснял ранее).
Эта константа записывается в регистр специального назначения **EEAdr**, который используется только при работе с EEPROM памятью данных.
Число, которое в него записывается, определяет адрес той ячейки, из которой будет производиться считывание.
Вот Вам и "место закладки" адреса ячейки, из которой будет производится считывание.
Итак, ячейка выбрана и далее нужно инициализировать чтение (разрешить считывание, из нее, числа).
Этим делом "рулит" нулевой бит регистра специального назначения **EECon1**, который находится в 1-м банке и используется только при работе с EEPROM памятью данных.
Следовательно, нужно перейти в 1-й банк, установить в **1** бит **№0** регистра **EECon1** и опять вернуться в 0-й банк, так как следующая команда (**movf EEData,W**) обращается к регистру, который "лежит" в 0-м банке.
После команды перехода в 0-й банк, располагается группа из двух команд.
После исполнения первой команды этой группы (**movf EEData,W**), предварительно записанное в ячейку, с адресом **02h**, число **64h** (**.100**), копируется в регистр **W** (аккумулятор), а после исполнения второй команды этой группы (**movwf Registr**), число **64h** (**.100**) копируется, из регистра **W**, в регистр оперативной памяти **Registr**, с содержимым которого (с числом **64h**), в дальнейшем, можно произвести ту или иную операцию.
Вот Вам и ответ на вопрос: "В какое место группы команд чтения данных, из EEPROM памяти данных, нужно вставить название регистра, в который считывается байт"?
Регистр специального назначения **EEData** (регистр данных) задействуется только при работе с EEPROM памятью данных.
Теперь, в соответствии с алгоритмом работы программы, нужно произвести операцию инкремента.
Итак, после исполнения команды **movwf Registr**, процедура чтения данных, из 3-й ячейки, закончилась и можно увеличить на **1** число **.100**, хранящееся в регистре **Registr**.
Это делается при помощи команды **incf Registr,F**.

После ее исполнения, в регистре **Registr** будет "лежать" число **.101** (**65h**), которое, в соответствии со сформулированным ранее алгоритмом работы программы, необходимо записать все в ту же, 3-ю ячейку EEPROM памяти данных.
Делаем это.
Разработчики ПИКов рекомендуют начинать процедуру записи с команды глобального запрета прерываний **bcf Intcon,GIE**.
Обратите внимание на то, что номер бита (7-го) регистра **Intcon** в команде не указан, так как в "шапке" программы, биту с номером 7, директивой **equ**, присвоено название **GIE**.
По этой причине, можно указывать не номер бита, а его название.
Можно эту команду оформить так: **bcf Intcon,7**.
В этом случае, из "шапки" программы, можно убрать строку **GIE equ 7**.
Глобальный запрет прерываний также можно осуществить при помощи байт-ориентированной команды **clrf Intcon**.
Если на момент начала исполнения одной из этих команд, в 7-м бите регистра **Intcon** ранее уже был записан **0**, то (по аналогии с командой **bcf Status,RP0**) эту команду можно исключить из текста "программы".
Формальная необходимость запрета прерываний обусловлена тем, что если прерывания по окончанию записи в EEPROM (см. **6**-й бит регистра **Intcon**) будут разрешены, то после окончания записи, рабочая точка программы "улетит" в ПП прерывания, которой, в данном случае, нет ("глюк").
Для недопущения этого, нужно просто глобально запретить все прерывания командой **bcf Intcon,GIE** (**bcf Intcon,7**).
Эта команда, в отличие от команды **clrf Intcon**, не "выключает" прерывания от других источников прерываний, а только (в интервале времени от исполнения команды **bcf Intcon,GIE** и до исполнения команды **bsf Intcon,GIE**) запрещает их.
После команды глобального запрета прерываний, следуют 2 команды адресного выбора ячейки.
Они в точности такие же, как и команды адресного выбора ячейки при чтении (задействована одна ячейка) и останавливаться на них я не буду.
После выполнения следующей пары команд, число **.101** (**65h**), через регистр **W**, копируется из регистра оперативной памяти **Registr**, в регистр специального назначения **EEData** и на этом, процесс предварительной подготовки к записи заканчивается.
Теперь нужно сначала разрешить, а затем и произвести запись.
Запись в выбранную ячейку EEPROM памяти данных, разрешается установкой в **1** второго бита регистра **EECon1**.
Он "лежит" в 1-м банке.
Следовательно, перед тем как это сделать, нужно выбрать 1-й банк (**bsf Status,RP0**).
После разрешения записи (**bsf EECon1,2**), можно произвести эту запись.
Для этого используется обязательная последовательность из пяти команд (см. текст примера).
При этом, производятся операции с так называемым "физически не реализованным" регистром **EECon2**, технические детали которых достаточно "туманны" (разработчики не объясняют).
Собственно говоря, программисту от этого "не тепло и не холодно", так как возможность "манёвра" начисто отсутствует, и по этой причине, ничего кроме этой обязательной последовательности команд, он и не сможет применить.
Вот и применяем ее.
После исполнения обязательной последовательности команд, флаг прерывания по окончанию записи в EEPROM устанавливается в **1** (флаг поднят).
Это и есть критерий окончания записи.
То есть, запись завершена и можно выйти из процедуры записи.
Но для того чтобы можно было осуществить следующую запись, этот флаг нужно программно ("принудительно") сбросить (**bcf EECon1,4**).
Это единственный вариант его сброса. Аппаратно (автоматически) он не сбрасывается.
Теперь можно перейти из 1-го банка в 0-й (**bcf Status,RP0**).
Вопрос: "Почему переход в 0-й банк осуществляется именно в этом месте, а не ранее"?
Ответ: потому, что предшествующие команды, начиная с команды **bsf EECon1,2**, работают с регистрами 1-го банка.

В тексте примера предполагается, что следующие, после команды **bcf Status,RP0**, операции будут производиться с регистрами 0-го банка (вероятность этого - большая), но если после завершения процедуры записи, следующие операции будут производиться с регистрами 1-го банка (вероятность этого - маленькая), то команду **bcf Status,RP0** нужно "вырезать" из концовки процедуры записи и перенести ниже (по тексту программы), "врезав" ее в то "место", в котором заканчиваются операции с регистрами 1-го банка и начинаются операции с регистрами 0-го банка.
Осталось только обратить внимание на "шапку" программы.
Рассмотрев ее, Вы сможете убедиться в том, что всё, относящееся к рассмотренной части программы, в ней "прописано" и ответ на вопрос "что откуда взялось?" должен быть Вам понятен.
Советую Вам переписать эти стандартные процедуры на листок и "приобщить" его к имеющимся у Вас распечаткам.
Примечание: в данном случае, рассмотрена процедура типа **чтение - модификация – запись**, в "границах" только одного полного цикла программы.
Не трудно догадаться, что произойдет в ходе исполнения последующих, полных циклов программы: с каждым циклом, значение числа, записываемого в 3-ю ячейку EEPROM памяти данных, будет увеличиваться на **1**.
До числа .255 включительно.
После этого, в ходе отработки следующего, полного цикла программы, произойдет переход от числа **.255** к числу **.0**. Затем к числу **.1**, **.2** и т.д.
Это называется **счетом по кольцу**.
Если применить операцию декремента, то все будет наоборот.
Для того чтобы можно было отследить изменения, я "модифицировал" нерабочую программу **eeprom.asm** таким образом, чтобы ассемблирование прошло успешно.

Эта "модифицированная" программа называется eeprom_a.asm
(находится в папке ***"Тексты программ"***).

```
;*******************************************************************************
;                          РАБОТА С EEPROM ПАМЯТЬЮ ДАННЫХ
;*******************************************************************************
;                                  "ШАПКА ПРОГРАММЫ"
;===============================================================================
;.............................................................
;===============================================================================
; Определение положения регистров специального назначения.
;===============================================================================
Intcon      equ         0Bh         ; Регистр Intcon.
Status      equ         03h         ; Регистр Status.
EEData      equ         08h         ; EEPROM - данные
EECon1      equ         08h         ; EECON1 - банк1.
EEAdr       equ         09h         ; EEPROM - адрес
EECon2      equ         09h         ; EECON2 - банк1.
;.............................................................
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
Registr     equ         0Ch         ; Регистр оперативной памяти, используемый
                                    ; при работе с EEPROM.
;.............................................................
;===============================================================================
; Определение места размещения результатов операций.
;===============================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;===============================================================================
; Присваивание битам названий.
;===============================================================================
RP0         equ         5           ; Бит выбора банка.
GIE         equ         7           ; Бит глобального разрешения прерываний.
;===============================================================================
```

```
            org         2100h       ; Обращение к EEPROM памяти данных.
            DE          0h,0h,64h   ; Записать в ячейки с адресами .0, .1, .2
                                    ; числа 0h, 0h, 64h (.100) соответственно.
            DE          0h,0h,0h    ; Записать в ячейки с адресами .3, .4, .5
                                    ; числа 0h, 0h, 0h соответственно.
            DE          "Korabelnikow E.A. Rus. Lipetsk. 03.2005"  ; Записать
                                    ; символы в ячейки с адресами с .6 по .46
;===============================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*******************************************************************************

;*******************************************************************************
;                                РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;*******************************************************************************
START                               ; .......................
;.........................................................
;===============================================================================
;                             Чтение данных из EEPROM
;===============================================================================
; Считывание содержимого байта с адресом 02h и запись его в регистр общего
; назначения Registr.
;-------------------------------------------------------------------------------
            bcf         Status,RP0  ; Переход в нулевой банк.

            movlw       2           ; Записать в регистр W константу 02h.
            movwf       EEAdr       ; Скопировать 02h, из регистра W,
                                    ; в регистр EEAdr.
            bsf         Status,RP0  ; Переход в первый банк.
            bsf         EECon1,0    ; Инициализировать чтение.
            bcf         Status,RP0  ; Переход в нулевой банк.

            movf        EEData,W    ; Скопировать число из ячейки EEPROM
                                    ; с адресом 02h, в регистр W.
            movwf       Registr     ; Скопировать число, из регистра W,
                                    ; в регистр Registr.
;-------------------------------------------------------------------------------
; Изменение содержимого регистра Registr.
;-------------------------------------------------------------------------------
            incf        Registr,F   ; Увеличить на 1 содержимое регистра Registr с
                                    ; сохранением результата в нем же.
;===============================================================================
;                            Запись данных в EEPROM.
;===============================================================================
; Запись содержимого регистра Registr в ячейку EEPROM с адресом 02h.
;-------------------------------------------------------------------------------
            bcf         Intcon,GIE  ; Глобальный запрет прерываний.

            movlw       2           ; Записать в регистр W константу 02h.
            movwf       EEAdr       ; Скопировать константу 02h, из регистра W,
                                    ; в регистр EEAdr.
            movf        Registr,W   ; Скопировать число, из регистра Registr, в
                                    ; регистр W.
            movwf       EEData      ; Скопировать число, из регистра W, в ячейку
                                    ; EEPROM с адресом 02h.
            bsf         Status,RP0  ; Переход в первый банк.
            bsf         EECon1,2    ; Разрешить запись.

            movlw       055h        ; Обязательная
            movwf       EECon2      ; процедура
            movlw       0AAh        ; при записи.
            movwf       EECon2      ; ----"----
            bsf         EECon1,1    ; ----"----

            bcf         EECon1,4    ; Сбросить флаг прерывания по окончании
                                    ; записи в EEPROM.
```

```
            bcf         Status,RP0  ; Переход в нулевой банк.
;-----------------------------------------------------------------
; Примечание: время исполнения полного цикла программы должно быть
; более времени, необходимого для окончания записи в EEPROM.
;-----------------------------------------------------------------
;.................................................................
            goto        START       ; Переход на новый цикл программы.
;*****************************************************************************
            end                     ; Конец программы.
```

В ней организован полный цикл программы и точками с запятыми заблокировано то, по поводу чего **MPLAB** "ругается".
Смысл примечания **время исполнения полного цикла программы должно быть более времени, необходимого для окончания записи в EEPROM**, будет раскрыт позднее.
Под программу **eeprom_a.asm**, создайте проект, загрузите ее в проект и произведите ассемблирование.
После этого, откройте окно ***EEPROM Window*** (***Window → EEPROM Memory***), назначьте точку остановки на команде **goto START** (на той, ниже которой расположена директива **end**) и сбросьте программу на начало.
А теперь от души пощелкайте по кнопке с **зеленым** светофором (или по клавише **F9**) и обратите внимание на ячейку **EEPROM** памяти с адресом **02h**.
Вы увидите, что с каждым таким "щелчком", ее содержимое будет увеличиваться на единицу.

В ПИКах предусмотрены меры по предотвращению случайной записи в EEPROM память данных, при сбоях в работе программы, снижении напряжения питания и других "гадостях".
Если бит защиты (см. биты конфигурации) установлен, то считать данные, из EEPROM памяти данных, с помощью программатора, теоретически, нельзя.
На работе программы, "зашитой" в ПИК, это не отражается.
При записи, теоретически, могут быть совершены ошибки, но практически, их вероятность мала.
В "особо ответственных" программах, задействующих EEPROM память данных, организуются проверки правильности записи.
По принципу сверки "эталона" (того, что должно записаться) с фактическим (считываемым после записи) результатом записи (так называемая верификация).
По результатам этой сверки, делается вывод о том, должна ли программа исполняться далее или нужно повторить запись до получения совпадения, а только после этого исполнять ее далее.
Это случай побайтной проверки.
В том случае, если требуется проверить правильность записи группы байтов ("массива" данных), то критерием безошибочности или ошибочности записи может быть так называемая "контрольная сумма".
Во многих случаях, контрольная сумма есть результат простейшего, последовательного суммирования по кольцу (для того чтобы сформировать однобайтный результат суммирования) всех числовых значений байтов "массива" данных.
Естественно, что для того чтобы было с чем сравнивать, до проверки на контрольную сумму, должен быть сформирован "эталон".
В случае обнаружения расхождения, производится перезапись всего этого "массива" данных, вплоть до устранения этого расхождения.
Можно применить и другие виды проверок, например, по признаку четности или нечетности.
Это, как говорится, личное дело каждого программиста.

13. Флаги. Работа с флагами. Как работает цифровой компаратор. Перенос и заем.

Ранее, во 2-м разделе, я упоминал о флагах.
Пришла пора с некоторыми из них разобраться.
Посмотрите в распечатки регистров специального назначения.
Названия битов флагов выделены **зеленым** цветом.
Естественно, что слово "флаг" нельзя воспринимать буквально. Это аналогия.
Факт поднятие флага "сигнализирует" о том, что произошло какое-то конкретное событие.
Факт "опущения" (сброса) флага "сигнализирует" о том, что этого конкретного события не произошло.
Каждый флаг "привязывается" к "своему" типу события.
Например, один флаг "реагирует" на факт наличия или отсутствия нулевого результата произведенной операции (флаг **Z**), другой флаг "реагирует" на факт наличия или отсутствия переполнения сторожевого таймера (флаг **-TO**), третий, на факт наличия или отсутствия переноса-заёма (флаг **C**) и т.д.
Исходя из этого, становится понятным и их практическое предназначение: анализируя состояния флагов, программист может осуществлять контроль за результатом исполнения задуманных им операций, совершаемых при помощи команд, к которым "привязаны" те или иные флаги (см. правый столбец распечатки команд).
В этом смысле, они удобны тем, что как бы выполняют функции "виртуального пробника", ведь программист не может "заглянуть" внутрь ПИКа и что-то в нем замерить, с использованием реального пробника.
С помощью флагов, программист может поставить ход исполнения программы в зависимость от фактов их поднятия или "опущения", что наиболее востребовано на практике.
Существуют **2 группы флагов**.
К первой группе относятся флаги, после поднятия которых, их не нужно программно сбрасывать.
Они "реагируют" и на наличие факта "привязанного" к ним события (флаг поднят), и на его отсутствие (флаг опущен).
Например, если с помощью команды **DECF** (но не с помощью команды DECFSZ, так как она не оказывает влияния на флаги) осуществляется декремент числа **.01**, то, после первого декремента (результат операции **.00**), флаг нулевого результата автоматически поднимется (бит **Z** установится в **1**), а после следующего декремента (результат операции **.255**), он автоматически сбросится (бит **Z** установится в **0**).
Ко второй группе относятся флаги, после поднятия которых, их нужно программно сбрасывать.
Проще говоря, один раз поднявшись, эти флаги не сбросятся до тех пор, пока программист их "принудительно" не сбросит программными средствами.
Например, в ранее рассмотренной программе **Retr_1.asm**, флаг внешнего прерывания по входу **INT** с названием **INTF**, при наличии внешнего прерывания по входу **INT**, устанавливается в **1** (поднимается).
Если его не сбросить (**0**) в интервале времени отработки ПП прерывания (до команды **retfie**), то после возврата из ПП прерывания, факт поднятия этого флага будет "восприниматься" ПИКом как наличие управляющего сигнала "ухода" в прерывание по входу **INT**, со всеми вытекающими из этого, "мерзопакостными" последствиями (работа программы нарушится).
Именно по этой причине, флаг **INTF**, до исполнения команды **retfie**, нужно обязательно сбросить.
Посмотрите в распечатки регистров специального назначения.
Те флаги, в комментариях к которым указано **"сбрасывается программно"**, относятся ко 2-й группе флагов, а те флаги, в комментариях к которым этого нет, относятся к 1-й группе флагов.
Таким образом, флаги 1-й группы можно назвать как бы "безобидными".
В том смысле, что если их состояния программно не анализируются (об этом будет рассказано далее), то на работу программы они не оказывают влияния.
Проще говоря, в этом случае, они "всем скопом" могут как угодно подниматься и опускаться.
Работа устройства от этого не нарушится.
Флаги 2-й группы не такие "безобидные".
За их состояниями нужно следить (учитывать при составлении программы).

Если такие флаги задействованы в программе, то после того как любой из них поднялся, его обязательно нужно программно сбросить.
Если этого не сделать, то "наказание" последует незамедлительно (нарушение работы программы).
Посмотрите в распечатку команд.
В правом столбце, с названием **"Флаги"**, Вы увидите стандартные названия флагов (битов флагов), на которые воздействует та или иная команда.
Есть команды, которые воздействуют сразу на несколько флагов (например, **ADDWF**), есть команды, которые воздействуют на один флаг (например, **CLRF**), а есть и команды, которые не воздействуют на флаги (например, **DECFSZ**).
Если Вы сравните общее количество различных флагов "прописанных" в распечатке команд и в распечатках регистров специального назначения, то Вы заметите, что в распечатке команд, их количество меньше, чем в распечатках регистров специального назначения.
Объяснение этому следующее.
В распечатках регистров специального назначения, Вы видите все флаги, а команды воздействуют только на их часть.
В правом столбце распечатки команд, Вы не найдете флагов 2-й группы.
Проанализировав содержимое этого столбца, Вы убедитесь в том, что наиболее часто встречается флаг нулевого результата **Z**.
По этой причине, он заслуживает особого внимания.
Возникает вопрос: "Могут ли флаги, кроме контрольных функций, выполнять какие-либо более значимые функции. Например, каким-то образом воздействовать на ход исполнения программы"?
Ответ: да, могут. И еще как могут**...**
Ранее уже говорилось о том, к чему может привести отсутствие программного сброса флагов 2-й группы.
Это есть не что иное, как прямое воздействие на ход исполнения программы.
Флаги 1-й группы также могут влиять на ход исполнения программы, но только несколько иным, косвенным образом.
Далее, будем работать с одним из самых "ходовых" флагов,
флагом нулевого результата Z.
Изначально, в обучающе-тренировочных целях, принцип работы с ним можно определить так: перед исполнением команды (эта команда должна воздействовать на флаг **Z**), флаг **Z** сбрасывается (**2**-й бит регистра **STATUS** устанавливается в **0**), а потом выполняется команда, с последующим анализом его (**Z**) состояния.
Сразу оговорюсь: такое определение принципа работы с флагом **Z** (и вообще, с флагами 1-й группы), совсем не является рациональным, но позволяет "вжиться/вживиться" в "механику" его работы, а это сейчас и есть самое главное.
Результат исполнения команды (число) может быть нулевым или не нулевым, то есть возможны 2 сценария.
Из этого следует то, что если проанализировать состояние флага **Z** после исполнения команды, которая влияет на состояние флага **Z**, то можно "разветвить" программу на 2 сценария.
Таким образом, ***выбор одного из двух сценариев дальнейшей работы программы ставится в зависимость от ответа на вопрос: "Каков результат исполнения команды: нулевой или не нулевой"?***
Анализ состояния флага **Z** (и других флагов тоже) осуществляется с помощью бит-ориентированных команд ветвления **btfsc** или **btfss**.
В результате исполнения любой из них, происходит ветвление на 2 сценария:
- в один из них, рабочая точка программы "уходит" при нулевом результате исполнения, команды (она должна воздействовать на флаг **Z**), которая предшествует команде ветвления,
- а в другой, при ненулевом результате исполнения той же команды.

Пример.
Допустим, что необходимо составить программу под устройство типа "реле времени" (например, для фотопечати).
То есть, имеется кнопка, после нажатия на которую, производится отсчет заданного интервала времени.
Таким образом, речь идет о счетчике, который, конечно же, можно реализовать так, как это

сделано в программах **Multi** или **cus**.
Но в нашем случае, нужно пофантазировать и "родить" вычитающий счетчик, работающий по принципу анализа состояния флага нулевого результата **Z**.
Фрагмент этого счетчика будет выглядеть так:

```
                .....................
XYZ             .....................
                .....................
                 bcf        Status,2
                 decf       ABC,F
                 btfss      Status,2
                 goto       XYZ
                .....................
                .....................
```

ABC - название регистра общего назначения,
XYZ - название циклической подпрограммы,
строки с точками - предыдущие и последующие команды программы.

Вы видите, что этот фрагмент "врезан" в программу, и по логике работы устройства, эта группа команд должна начать работать только после нажатия на кнопку запуска.
Пока кнопка не нажата, рабочая точка программы должна "крутиться" где-то в программе, "обходя" указанную выше группу команд, и при этом, состояние кнопки запуска должно опрашиваться (состояние ожидания нажатия кнопки).
После нажатия на кнопку и последующего, очередного опроса клавиатуры, факт ее нажатия будет обнаружен, и рабочая точка программы, на некоторое время, "закольцуется" в циклической подпрограмме, "поставленной на счетчик" (задержка).
Указанная выше группа команд, должна входить в состав этой циклической подпрограммы "на правах отфутболивателя рабочей точки".
При таком "раскладе", рабочая точка программы может выйти из циклической подпрограммы только при условии, что флаг нулевого результата поднялся (**Z**=**1**).
Итак, после нажатия на кнопку, рабочая точка программы "рано или поздно" установится на команду **bcf Status,2** (сброс флага **Z**).
Следующей исполняется команда декремента **decf ABC,F**, воздействующая на флаг **Z**.
В регистре общего назначения, условно названном **ABC**, на момент начала счета, находится какое-то предварительно (до входа в циклическую ПП) записанное в него число (константа), например **.10** (количество отрабатываемых циклов).
После декремента, число **.10** уменьшается до **.09** и сохраняется в том же регистре **ABC**.
Далее, с помощью команды **btfss Status,2**, производится анализ состояния флага **Z**.
Если результат декремента не нулевой (**Z**=**0**), то произойдет переход на начало циклической подпрограммы (**goto XYZ**) и все повторится снова.
В конце каждого такого "витка", содержимое регистра **ABC** будет уменьшаться на 1.
"Рано или поздно" (на 10-м "витке"), результат декремента станет нулевым (**Z**=**1**), после чего произойдет переход в сценарий типа "программа исполняется далее" (выход из циклической подпрограммы).
После этого, рабочая точка программы будет "гонять по другим кольцам" полного цикла программы, обходя указанную выше, циклическую подпрограмму и осуществляя слежение за состоянием кнопки.
Время задержки будет равно времени нахождения рабочей точкой программы внутри циклической подпрограммы.
Изменяя числовое значение константы, можно изменить продолжительность этой задержки.
По сравнению с тем, что рассматривалось ранее (применялась команда **decfsz**), такой способ "постановки на счетчик", не является выигрышным (большее количество команд).
Я о нем рассказал только для того, чтобы "проиллюстрировать" работу с флагом **Z**.
Следующий пример работы с эти флагом более востребован.
Задействуя флаг **Z**, можно реализовать такое нужное устройство, как цифровой компаратор, то есть, устройство, осуществляющее анализ "входного" числа на предмет его "попадания" (или нет) в определенную "зону" чисел.
Если анализируемое число не попадает в эту "зону", то исполняется один сценарий работы программы, а если попадает, то другой.

В состав такой "зоны" чисел, в зависимости от желания программиста, могут входить от двух чисел и более.
Цифровой компаратор может быть частью, например, системы фазовой автоподстройки частоты, дешифратора, системы блокировки и т. д., то есть, весьма обширного класса устройств.

Файл с примером реализации цифрового компаратора, с формализованными комментариями, называется flag.asm (находится в папке ***"Тексты программ"***).

Это выглядит так:

```
;*******************************************************************************
; flag.asm      ПРИМЕР ИСПОЛЬЗОВАНИЯ ФЛАГА НУЛЕВОГО РЕЗУЛЬТАТА "Z"
;*******************************************************************************
Вопрос: что должно произойти?
Ответ: если, на момент анализа содержимого регистра буферной памяти, в нем
записано любое из чисел от .00 до .04 включительно, то рабочая точка программы
должна "уйти" в один сценарий работы "программы" (в подпрограмму YES), а если
указанное  выше число не входит в  "зону" чисел от .00 до .04 включительно, то
рабочая точка программы должна "уйти" в другой сценарий работы "программы" (в
подпрограмму NO).

;*******************************************************************************
;                                   "ШАПКА ПРОГРАММЫ"
;===============================================================================
;.....................................................
;.....................................................
;===============================================================================
; Определение положения регистров специального назначения.
;===============================================================================
Status      equ         03h         ; Регистр Status.
;.....................................................
;.....................................................
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
Registr     equ         0Ch         ; Регистр буферной памяти.
;.....................................................
;.....................................................
;===============================================================================
; Определение места размещения результатов операций.
;===============================================================================
W           equ         0           ; Результат направить в аккумулятор.
;.....................................................
;===============================================================================
; Присваивание битам названий.
;===============================================================================
RP0         equ         5           ; Бит выбора банка.
Z           equ         2           ; Бит флага нулевого результата.
;.....................................................
;===============================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*******************************************************************************

;*******************************************************************************
;                            РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;*******************************************************************************
START       .....................................
            .....................................
            .....................................
;-------------------------------------------------------------------------------
; ПП NO, в которую "уходит" рабочая точка программы, если анализируемое число не
; попадает в "зону" чисел от .00 до .04 включительно.
```

```
;-------------------------------------------------------------------------------
NO          ....................................
            ....................................
;-------------------------------------------------------------------------------
            ....................................
            ....................................
;===============================================================================
; Реализация цифрового компаратора.
; Решение задачи о вхождении или нет анализируемого числа в "зону" чисел от .00
; до .04 включительно.
;===============================================================================
            movf        Registr,W   ; Скопировать содержимое регистра Registr
                                    ; в регистр W.
            bcf         Status,Z    ; Сбросить флаг нулевого результата Z.

            sublw       .00         ; Вычесть из константы .00 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; Проверка состояния флага нулевого
                                    ; результата (Z).
            goto        YES         ; Если Z=1 (нулевой результат операции),
                                    ; то "уход" в ПП YES.
            movf        Registr,W   ; Если Z=0 (ненулевой результат операции), то
                                    ; содержимое регистра Registr копируется
                                    ; в регистр W.

            sublw       .01         ; Вычесть из константы .01 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; --------------"--------------
            goto        YES         ; --------------"--------------
            movf        Registr,W   ; --------------"--------------
                                    ; --------------"--------------

            sublw       .02         ; Вычесть из константы .02 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; --------------"--------------
            goto        YES         ; --------------"--------------
            movf        Registr,W   ; --------------"--------------
                                    ; --------------"--------------

            sublw       .03         ; Вычесть из константы .03 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; --------------"--------------
            goto        YES         ; --------------"--------------
            movf        Registr,W   ; --------------"--------------
                                    ; --------------"--------------

            sublw       .04         ; Вычесть из константы .04 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; Проверка состояния флага нулевого
                                    ; результата (Z).
            goto        YES         ; Если Z=1 (нулевой результат операции),
                                    ; то "уход" в ПП YES.
            goto        NO          ; Если Z=0 (ненулевой результат операции),
                                    ; то происходит безусловный переход в ПП NO.
;-------------------------------------------------------------------------------
; ПП YES, в которую "уходит" рабочая точка программы, если анализируемое число
; попадает в "зону" чисел от .00 до .04 включительно.
;-------------------------------------------------------------------------------
YES         ....................................
            ....................................
;-------------------------------------------------------------------------------
            ....................................
            ....................................
;*******************************************************************************
            end                     ; Конец программы.
```

Файл с примером реализации цифрового компаратора, с неформализованными комментариями, называется flag_1.asm (находится в папке ***"Тексты программ"***).

Это выглядит так:

```
;*******************************************************************************
; flag_1.asm     ПРИМЕР ИСПОЛЬЗОВАНИЯ ФЛАГА НУЛЕВОГО РЕЗУЛЬТАТА "Z"
;*******************************************************************************
;.....................................................
;.....................................................
;.....................................................
;.....................................................
;===============================================================================
; Реализация цифрового компаратора.
; Решение задачи о вхождении или нет анализируемого числа в "зону" чисел от .00
; до .04 включительно.
;===============================================================================
            movf        Registr,W   ; Скопировать содержимое регистра Registr
                                    ; в регистр W.
            bcf         Status,Z    ; Сбросить флаг нулевого результата Z.

            sublw       .00         ; Вычесть из константы .00 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; Результат  вычитания нулевой?
            goto        YES         ; Да. --->"уход" в ПП YES.
            movf        Registr,W   ; Нет. --->скопировать содержимое регистра
                                    ; Registr в регистр W.
            sublw       .01         ; Вычесть из константы .01 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; Результат  вычитания нулевой?
            goto        YES         ; Да. --->"уход" в ПП YES.
            movf        Registr,W   ; Нет. --->скопировать содержимое
                                    ; регистра Registr в регистр W.
            sublw       .02         ; Вычесть из константы .02 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; Результат  вычитания нулевой?
            goto        YES         ; Да. --->"уход" в ПП YES.
            movf        Registr,W   ; Нет. --->скопировать содержимое регистра
                                    ; Registr в регистр W.
            sublw       .03         ; Вычесть из константы .03 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; Результат  вычитания нулевой?
            goto        YES         ; Да. --->"уход" в ПП YES.
            movf        Registr,W   ; Нет. --->скопировать содержимое регистра
                                    ;  Registr в регистр W.
            sublw       .04         ; Вычесть из константы .04 содержимое
                                    ; регистра W.
            btfsc       Status,Z    ; Результат  вычитания нулевой?
            goto        YES         ; Да. --->"уход" в ПП YES.
            goto        NO          ; Нет. --->"уход" в ПП NO.
;-------------------------------------------------------------------------------
            ....................................
            ....................................
```

Проект под них в **MPLAB** создавать не нужно. Просто откройте (***File*** → ***Open***).
Вы видите пример программы, реализующей цифровой компаратор.
Сначала, в общем виде, проанализируем принцип ее работы.
Цифровой компаратор анализирует "поступающие на его вход" числа, на предмет их попадания или нет в "зону" чисел от **.00** до **.04** включительно.
"Ширину" этой "зоны" можно сделать другой. Можно также и сместить ее по числовой оси.
При соответствующем наращивании количества команд, можно расширить эту "зону", а при сокращении, уменьшить.
В результате "протаскивания" рабочей точки программы через цифровой компаратор, она должна "неизбежно уйти" в один из двух сценариев работы программы.

В данном случае, один сценарий будет исполняться после безусловного перехода рабочей точки программы в ПП **YES**, а другой, в ПП **NO**.
В них может "делаться все что угодно". Это зависит от замысла программы.
Названия этих подпрограмм условны и их можно назвать иначе.
Если переходы осуществляются не на первые команды подпрограмм, то речь идет о метках.
Я расположил ПП **NO** вверху, а ПП **YES** внизу (по тексту программы), но они могут располагаться и иначе.
В качестве элемента оперативной памяти, используется регистр **Registr** (см. "шапку" программы).
На момент начала исполнения первой команды группы команд цифрового компаратора, в регистре **Registr** должно "лежать" число, величину которого нужно "заанализировать".
В данном, учебно-тренировочном случае, не важно, каким именно образом "возникло" это число (существует множество вариантов "возникновения").
Приглядевшись к командам, реализующим цифровой компаратор, Вы должны заметить аналогию с рассмотренным выше примером.
Так как командой, результат исполнения которой анализируется, является команда операции с константой, предполагающая задействование регистра **W** (команда **sublw**, воздействующая на флаг **Z**), то перед ее исполнением, необходимо скопировать в регистр **W** число, "находящееся" в регистре **Registr**, что и делается с помощью команды **movf Registr,W**.
Перед командой, результат исполнения которой будет анализироваться (**sublw**), сбрасываем (опускаем) флаг **Z**.
Не из-за того, что это необходимо (он относится к флагам 1-й группы и его можно не сбрасывать), а для того, чтобы "прочувствовать сам процесс".
Такого рода навыки будут архивостребованы при работе с флагами 2-й группы.
Результат исполнения команды **sublw .00** будет нулевым только тогда, когда содержимое регистра **Registr** также будет нулевым (0-0=**0**).
При этом, флаг **Z** будет поднят (**Z**=**1**).
Во всех остальных случаях, результатом исполнения команды **sublw .00** будет число не равное нулю.
При этом, флаг **Z** будет опущен (**Z**=**0**).
После исполнения команды **sublw .00**, результат ее исполнения анализируется командой **btfsc Status,Z** (анализ состояния флага **Z)**.
Если в регистре **Registr** "лежит" число **.00** (**Z**=**1**), то рабочая точка программы "улетает" в ПП **YES**.
Во всех остальных случаях (**Z**=**0**), она продолжает свое движение по сценарию "программа исполняется далее".
То есть, начинается отработка следующей, "однотипной" группы команд анализа, но только с использованием числа .**01**.
В ходе исполнения этой группы команд, выражаясь неформально, дается ответ на вопрос: "Анализируемое число равно числу **.01** или не равно?"
Если равно, то рабочая точка программы "летает" все в ту же ПП **YES**, а если не равно, то начинается отработка следующей, "однотипной" группы команд анализа, но только с использованием числа .**02**.
И так далее. До числа **.04** включительно.
Если не в одной из этих групп команд не произошел "улёт" рабочей точки программы в ПП **YES** (числовых соответствий не обнаружено), то после отработки группы команд анализа на соответствие числу **.04**, начнется отработка ПП **NO**.
В ПП **YES** и **NO**, примитивно выражаясь, "может делаться все, что угодно".
Если по факту "срабатывания" компаратора, нужно банально включить какое-то внешнее устройство, то команды **goto YES** можно заменить, например, на команды **bsf PortA,X** (**X** – номер бита).
Если во всех командах **bsf PortA,X**, номер бита один и тот же (задействован один вывод порта), то устройство будет включаться в диапазоне чисел.
Если номера этих битов разные (задействованы несколько выводов порта), то можно включать несколько устройств.
При этом, факт включения любого из них, будет "привязан" не к диапазону чисел, а к одному из его поддиапазонов или даже к конкретному числу (от **.00** до **.255**).
Могут быть и комбинации.

С помощью цифрового компаратора, по факту возникновения числового соответствия (или несоответствия), можно "запустить в работу" какую-то процедуру (например, процедуру вычисления), косвенно влияющую на управление "периферией".
Могут быть и другие варианты, здесь "есть где развернуться".
В качестве примера практического использования цифрового компаратора можно привести пример реализации (в упрощенном, общем виде), допустим, устройства пожарной сигнализации, реагирующего на температуру.
Предположим, что имеется термодатчик, который работает в диапазоне от 0 до 100 градусов Цельсия.
Необходимо, чтобы в диапазоне температур от 80 до 90 градусов, сработала некая "предупреждалка" (например, сирена), а в диапазоне температур от 91 до 100 градусов, включилась система автоматического пожаротушения.
Предположим, что датчик линейный, и изменению температуры от 0 до 100 градусов соответствует изменение напряжения на его выходе от 0 до 10в.
Предположим, что аналоговый сигнал датчика переведен в цифровую форму, и изменению напряжения в диапазоне 0 …10в линейно соответствует изменение значений чисел в диапазоне .0 … .100.
В этом случае, в команды первой группы проверок, должны быть "заложены" константы с числовыми значениями от .80 до .90, а в команды второй группы проверок - от .91 до .100.
Внутри такого "составного" компаратора, можно либо поменять местами эти группы, либо, в пределах группы, "перетасовать числа как колоду карт".
На работе устройства это не отразится.
А теперь все зависит от степени "навороченности" замысла конструктора.
Если она низкая, то можно подключить вход исполнительного устройства к какому-нибудь выводу порта, настроенному на работу "на выход", и с помощью команд **bcf** / **bsf**, программно управлять его состояниями.
Если она высокая, то используя команды переходов, нужно "уйти" в соответствующую подпрограмму (подпрограммы), так как в этом случае, потребуется большее количество команд.
Например, требуется, чтобы исполнительное устройство сработало по совокупности событий: высокие температура и задымленность.
Если первой анализируется температура, то при наличии числа, например, от .80 до .90, осуществляется переход на исполнение подпрограммы (подпрограмм) анализа задымленности.
В ее состав должна входить, аналогичная рассмотренной, группа команд цифрового компаратора, работающая с числами, величины которых зависят от уровня выходного сигнала датчика задымленности.
Если получены утвердительные ответы на оба вопроса ("Температура выше нормы?" и "Уровень задымленности выше нормы?"), то срабатывает исполнительное устройство пожаротушения.
Естественно, что при этом, конструкция и устройства и программы усложняется (два датчика).
Этот пример, может быть, и не слишком удачный, но "глобальный" смысл, я надеюсь, понятен.
Существуют и комбинированные (не "однообразные") варианты "ухода" рабочей точки программы, из группы команд, реализующей цифровой компаратор.
Например, если анализируемое число равно одной константе, то осуществляется прямое управление выводом порта, а если анализируемое число равно другой константе, то осуществляется переход на нечто такое, что сложнее прямого управления выводом порта.
В основном, цифровой компаратор является всего-лишь одним из элементов "программного комплекса", и этот "элемент" участвует в управлении исполнительным устройством (устройствами), состояния которого многовариантны.
В этом случае, речь идет не о прямом, а об опосредованном воздействии на исполнительное устройство.
Пример исполнительного устройства, состояния которого многовариантны → 7-сегментный индикатор.
Пример опосредованного воздействия цифрового компаратора, на работу 7-сегментных индикаторов → гашение незначащих нулей в линейке из нескольких таких индикаторов.

В очень общем виде, это выглядит так.
Несколько последовательно соединенных компараторов, анализируют несколько чисел, предназначенных для отображения в линейке, начиная от числа, отображаемого в самом левом (старшем) знакоместе линейки.
Если, до первой значащей цифры (не ноль), обнаружен ноль (нули), то он (они) гасится (не выводится на индикацию).
Правее первой значащей цифры (включая и ее), гашения нет.
"Фундаментом всего этого сооружения" и других подобных устройств, является цифровой компаратор, а его "сердцем", флаг нулевого результата **Z**, так что он вполне заслуживает к себе внимания.
Следует отметить то, что цифровой компаратор можно реализовать и с использованием флага **C**, речь о котором пойдет ниже.

Разбираемся с флагами C и DC.

Если принцип работы флага **Z** не является чем-то уж очень сложным, то понимание принципов работы флагов переноса-заёма **C** и десятичного переноса-заёма **DC**, для большинства начинающих, является своеобразным "камнем преткновения".
Постараюсь убедить Вас в том, что "не так страшен черт, как его малюют", а заодно, на примере флагов **C** и **DC**, расскажу о методике "въезда" в операции, которые вызывают затруднения.
Это выглядит так.
Создается некая группа команд, "обслуживающая" выполнение операции, вызывающей затруднения.
Для того чтобы можно было "полноценно" работать в симуляторе, необходимо "врезать" эту группу команд в любую простенькую программу, в которой регистр (регистры) специального назначения, использующийся в этой группе команд, "прописан" в "шапке" программы или "прописать" его, если такого регистра нет.
В противном случае, после ассемблирования, будет выдаваться сообщение об ошибке типа "происходит обращение к регистру, который не задействован".
Для того чтобы долго до нее не "добираться", эту группу команд целесообразно "врезать" в начале программы.
При этом, то, как именно будет работать программа, совершенно безразлично.
Все внимание - на "врезанную" группу команд.
В соответствии со сказанным, нужно как бы "родить дитя от суррогатной матери".
В качестве таковой, используем известную Вам, простенькую программу **Multi.asm**.
Откройте проект **Multi**.
Куда "врезать" проверочную группу команд?
Можно "вставить" ее в самое начало программы, но при этом нужно будет переносить название ПП **START** на первую команду этой группы.
Можно сделать и так, но лучше "оставить в покое" название подпрограммы, а "врезаться" где-нибудь пониже.
В "промежутке" между командами выбора 1-го банка и выбора 0-го банка, эту "врезку" можно делать только в том случае, если проверочная группа команд работает в 1-м банке.
В большинстве случаев, она работает в 0-м банке.
"Не мудрствуя лукаво", "врезаем" проверочную группу команд в "промежуток" между командами **bcf Status,5** и **movlw .32** ("и волки сыты, и овцы целы").
Теперь определяемся с ее составом.
Если необходимо разобраться с флагами **C** и **DC**, то нужно работать с командой, которая воздействует на оба этих флага.
Посмотрите в распечатку команд. Таких команд 4: две - сложения и две - вычитания.
Операция сложения, на мой взгляд, более проста и привычна, чем операция вычитания.
Значит, "отсеиваем" 2 команды вычитания.
Остались команды сложения **ADDWF** и **ADDLW**.
Можно применить и ту, и другую, но для обеспечения простоты "конструкции" проверочной группы команд, выгоднее использовать команду операции с константой (**ADDLW**), а не команду операции с содержимым регистра (**ADDWF**), так как в последнем случае, нужно будет задействовать регистр общего назначения, и запись в него производить в 2 приема (через регистр **W**).

То есть, в случае применения команды **ADDWF**, потребуется 5 команд, плюс задействование регистра общего назначения, а в случае применения команды **ADDLW**, потребуется всего 3 команды и никаких регистров задействовать не нужно.
Итак, используем команду **ADDLW** (суммирование содержимого регистра **W** и константы, с сохранением результата в регистре **W**).
Итак, первое слагаемое (константа) в наличии.
Ее числовое значение указывается в рабочей части команды **ADDLW**.
На момент исполнения команды **ADDLW**, второе слагаемое должно "лежать" в регистре **W**.
Следовательно, до исполнения команды **ADDLW**, в регистр **W**, нужно записать числовое значение второго слагаемого.
Для того чтобы, в симуляторе, выяснить, когда поднимаются флаги **C** и **DC**, перед выполнением команды **ADDLW** их необходимо сбрасывать (устанавливать в ноль).
Следовательно, перед исполнением команды **ADDLW**, должна исполниться команда **bcf Status,0**, если работаем с флагом **C**, или команда **bcf Status,1**, если работаем с флагом **DC**.
Примечание: команды **bcf** и **bsf** на флаги не влияют.
Получается следующее:
для работы с флагом **C**

```
movlw     X
bcf       Status,0
addlw     Y
```

для работы с флагом **DC**

```
movlw     X
bcf       Status,1
addlw     Y
```

X и **Y** – слагаемые числа.

Разбираемся с флагом C.

"Врежьте" указанную выше группу команд (для работы с флагом **C**), в текст программы **Multi.asm**, между командами **bcf Status,5** и **movlw .32**.
Естественно, **X** и **Y** "пропечатывать" не нужно. Это символьные обозначения.
Просто оставьте пустое место для чисел. Их можно будет "вписать" позднее.
Все готово для того чтобы "выпытать" у флага **C** его "тайны".
Методика этой "пытки" очень проста: в качестве чисел **X** и **Y** подставляем различные их значения (от **.00** до **.255**), смотрим, что из этого получится, и анализируем результат.
Сразу возникает вопрос: "Куда смотреть и какие значения чисел подставлять?"
Ответ: смотреть - в окно **RAM** или **SFR**, а из чего исходить при выборе значений этих чисел, давайте разбираться.
Посмотрите в комментарий к флагу **C**.
В нем говорится о каком-то старшем бите и не оговаривается, старшем бите чего?
Полубайт (4 бита) отпадает, так как с ним работает флаг **DC** (см. распечатку).
Остается байт.
Следовательно, речь идет о каком-то непонятном (пока) переносе, чего-то еще более непонятного, из старшего бита байта.
Байт может отобразить значения чисел от **.00** до **.255**, а старший бит байта устанавливается в **1** во второй половине этого диапазона чисел (начиная с **.128**) и сбрасывается, из **1**, в **0**, при переходе от числа **.255** к **.00**.
Вполне логично "привязаться" к этому, наиболее "резкому цифровому скачку" (переход от **.255** к **.00**) и предположить, что при этом, должно что-то произойти.
А именно, флаг **C** должен, на этот "скачек", "среагировать", то есть, подняться (установиться в **1**).
Вопрос: "В каком случае, результат исполнения команды **ADDLW** может быть бОльшим, чем **.255** ?"
Ответ однозначен: если сумма этих чисел больше, чем **.255**.
Но 8-битное число не может быть большим, чем **.255**.
А если нужно просуммировать, например .250 и .200 ?
Получится не .450, а (.250+.200)-.256=.194.

Число .256 "исчезло" по причине того, что разрядности одного регистра маловато будет.
В подобных случаях, если не принять мер по увеличению разрядности регистра, то в каждом случае превышения значения суммы, над числом .255, будет "исчезать" число, равное 256 (общее количество чисел, которые способен отобразить один байт).
В ПИКах, нельзя реализовать, например, 9, 10, 11, ... - разрядный регистр общего назначения.
Можно реализовать только разрядность, кратную 8-ми, задействуя несколько регистров общего назначения.
Только в этом случае можно работать с "большими" числами.
Количество задействованных регистров зависит от максимального значения числа, которое нужно в них отобразить.
Например, если это значение равно .50 000, то нужно использовать двухбайтный регистр (задействуются 2 регистра общего назначения), который способен отобразить числа от **.0** до (256x256)-1=**.65535**.
Итак, возникла необходимость в наращивании разрядности.
Вопрос: "Как программно "состыковать" два отдельных регистра"?
Ответ: начну "из аппаратного далекА".
Например, нужно нарастить разрядность двоичного счетчика 555ИЕ5.
Выход старшего разряда этого счетчика подключается к счетному входу еще одного счетчика, и этот второй счетчик (старший) начинает отсчитывать количество полных циклов счета первого (младшего) счетчика.
Так называемым выходом **переноса**, в данном случае, является выход триггера старшего разряда младшего по разрядности счетчика, а сигналом переноса является перепад напряжения, формируемый при переходе от числа .15 к числу .00 (555ИЕ5 работает с полубайтом).
А ведь регистры ПИКа имеют точно такую же основу (триггеры), что и 555ИЕ5 (и многие другие), следовательно, нарастить разрядность этих регистров можно по такому же принципу.
То есть, по принципу переноса, из старшего, двоичного разряда "младшей считалки", в младший, двоичный разряд "старшей считалки".
Если же речь идет о ПИКах, то задача формулируется так: ***нужно организовать перенос из старшего байта регистра младшего разряда в младший байт регистра старшего разряда***.
Только имеется большое горе: внутрь ПИКа с паяльником не залезешь и ничего не припаяешь. И вообще, гиблое это дело …
Но то, что выделено ***темно-синим цветом***, можно реализовать программно.
Чувствуете, "куда ветер дует?"
А "ветер дует" на флаг **C**.
В его названии есть слово "перенос" и "следит" он за байтом.
А что если флаг C поднимается в том случае, если результат суммирования больше, чем число .255 ?
Самостоятельно проверьте это "гениальное предположение", используя "вышележащую", проверочную группу команд (для флага **C**).
Изменяя значения чисел **X** и **Y** и производя, после этого, суммирования, обратите внимание (в окне **RAM**, а можно и в окне **SFR**) на состояния флага **C** (бит **№0** регистра **STATUS**, адрес регистра **03h**), в случаях, если сумма этих чисел меньше **.255** или больше **.255**.
Работайте в пошаговом режиме.
Не забывайте о том, что после изменения значений чисел **X** и/или **Y** (при этом, в текст программы вносятся изменения), нужно производить ассемблирование, а также и о том, что если речь идет об одном байте, то констант, бОльших чем **.255**, не бывает.
Полчаса такой работы более эффективны, чем несколько часов теории.
Обычно, после такого рода работы, наступает значительное "просветление", вплоть до того, что человек даже может удивиться, насколько простым оказалось то, что ранее вызывало затруднения.
Итак, что выясняется?
Выясняется следующее:

1. ***Если арифметический*** (а не по "кольцу") ***результат сложения двух чисел больше, чем число*** **.255**, ***то флаг C устанавливается в*** **1**. ***То есть, осуществлен*** **перенос**.
2. ***Если этот результат находится в числовом диапазоне*** **.00 … .255**, ***то флаг*** **C**

***устанавливается в* 0. *То есть,* переноса нет.**

***Вывод:* если по фактам возникновения переносов, инкрементировать содержимое регистра более старшего разряда, то можно безошибочно суммировать "большие" (условно) числа.**

До сих пор, речь шла об операции сложения.
Давайте разберемся с операцией вычитания.
Если в проверочной группе команд, Вы замените команду сложения **ADDLW**, на команду вычитания **SUBLW** (а также замените команду **bcf Status,0** на **bsf Status,0**. Почему? Поймете позже) и произведёте, аналогичную описанной выше (с учетом того, что результат может быть отрицательным), проверку, то выясните следующее:

1. ***Если арифметический результат вычитания двух чисел является числом положительным,*** (находится в числовом диапазоне **.00 … .255**)***, то флаг C устанавливается в* 1.**
2. ***Если этот результат меньше нуля, то есть, он является отрицательным, то флаг C устанавливается в* 0.**

Теперь, речь идет о заёме.
В первом случае, заёма нет.
Во втором случае, заём есть.
Вот Вам и ответ на вопрос: "Откуда, в комментарии к флагу **C**, взялось словосочетание **перенос - заем**"?

Обязательно следует уточнить, что **перенос** (при суммировании) или **заём** (при вычитании) возникает **не во всех случаях** суммирования или вычитания, а только в тех случаях, когда результат суммирования или вычитания "выходит за рамки" числового диапазона **.00 … .255** (с того или иного "края").
То есть, в тех случаях, ***когда арифметический результат суммирования больше числа .255 (перенос), и когда арифметический результат вычитания является отрицательным числом (заём).***
Во всех остальных случаях, речь идет просто о суммировании и просто о вычитании (без "прибамбасов").
Примеры:
- Если суммируются числа, например, **.254** и **.03**, то фактической суммой будет являться не число **.257** (в одном регистре отобразить его невозможно), а число **.01**, плюс **перенос** (флаг **C** установился в **1**).
- Если из числа **.01** вычитается число **.03**, то фактическим результатом вычитания не будет являться отрицательное число. Им будет являться число **.254**, плюс **заём** (флаг **C** установился в **0**).

Если Вы сравните эти примеры, то поймете, почему **заем имеет инверсное значение** (см. комментарии к флагу **C**).
И в самом деле, ***при переносе, бит флага C устанавливается в* 1*, а при заёме, он устанавливается в* 0**.
То есть, заём, по отношению к переносу, является действием типа "наоборот".
А теперь давайте разберемся с "легким бардачком", возникающим при этом, а заодно, внесем ясность и в вопрос о том, **обязательно ли нужно сбрасывать флаги 1-й группы (флаги Z, C, DC относятся к 1-й группе) или нет, перед выполнением команд, которые на эти флаги воздействуют?**
Если с переносом все понятно (бит **C** установился в **1**, то есть, флаг поднялся), то при заёме, флаг **C**, формально (так как мы ранее "привязались" к тому, что поднятию флага соответствует **1**), опустился.
"Виной этому" то, что **заем имеет инверсное значение**.
В этом случае (при применении команд **SUBWF** или **SUBLW**), нужно просто "проинвертировать привязку", то есть, изменить критерий "поднятия": флаг поднят → **0**, флаг опущен → **1**.
В соответствии с этим, вносим коррективы в проверочную группу команд (ранее предполагалось, что флаг **C** устанавливается в 1 и при переносе, и при заёме).

Если производится суммирование, то нужно применить команду **bcf Status,0**, а если вычитание, то нужно применить команду **bsf Status,0**.
В этом случае, на момент исполнения команд **ADDLW** или **SUBLW** (с учетом сформулированных выше критериев "поднятия"), флаг **C** будет гарантированно опущен.
Что касается ответа на вопрос, сформулированный выше (он выделен **темно-зеленым** цветом), то ответ такой: нет, не обязательно, но для начинающих, желательно (я об этом уже говорил ранее).
Прочитайте определение флагов 1-й группы, которое дано выше.
Это и есть первая часть ответа.
И в самом деле, зачем предварительно сбрасывать все те же флаги **Z, C, DC**, если их состояния полностью определяются результатами исполнения соответствующих команд.
Проще говоря, в каком бы из двух состояний не находился, до момента исполнения команды (той, которая на него воздействует), флаг 1-й группы, после ее исполнения, этот флаг будет установлен в то состояние, которое соответствует результату выполненной операции.
Таким образом, из группы команд, реализующих цифровой компаратор, и из группы проверочных команд, работающих с флагами **C** и **DC**, все команды сброса флагов можно смело убрать.
Вопрос: "Так что же, в конце концов, делать-то, убирать или не убирать"?
Ответ: если информация, изложенная выше, Вам предельно ясна и понятна, и Вы имеете хотя бы небольшой опыт составления и отладки программ, то команды предварительных сбросов флагов можно не применять, а если у Вас с этим проблемы, то отказываться от этих сбросов не стОит.
По причине того, что в последнем случае, процесс отладки программ (в части касающейся отладки подпрограмм, в которых происходят опросы состояний флагов 1-й группы) становится более наглядным.
Пример.
Предварительного сброса флага **C** нет.
В результате исполнения предшествующих команд программы, на момент исполнения команды **ADDLW**, бит флага **C** установился в **1**.
Допустим, суммируем числа **.100** и **.200**.
Значит, после исполнения команды **ADDLW**, бит флага **C** должен установиться в **1**.
Но в нем и так "выставлена" **1**.
В итоге, после исполнения команды **ADDLW**, состояние бита флага **C** не изменится (подтверждение ранее установленного состояния).
Работа программы не нарушится, но в симуляторе, Вы не увидите изменения состояния бита флага **C**, что для начинающих, которые недостаточно хорошо ориентируются во "флаговых делах", совсем не комфортно.
"На этой почве", могут возникнуть сомнения, на устранение которых будут потрачены силы и нервы.
При наличии такого рода затруднений, например, при суммировании, целесообразно предварительно установить бит флага **C** в ноль (**bcf Status,0**) так, как это сделано в группе проверочных команд, после чего можно будет с комфортом наблюдать за поднятием флага в окнах **RAM** или **SFR**.
Если нужно проконтролировать случаи сбросов флага, то перед исполнением соответствующей команды, можно либо вообще ничего не "врезать" (если "на влёте", флаг поднят), либо (если "на влёте", флаг опущен) предварительно поднять флаг **C** командой **bsf Status,0** (по принципу типа "хуже от этого не будет и изменения увижу").
Подобного рода "манипуляции" есть элемент удобства отслеживания работы программы.
Можно даже сказать, что "уловка".
Может быть, опытные программисты и посчитают такое внимание к контрольным (второстепенным) операциям излишним, но я так не считаю.
По той причине, что опытные программисты это знают, а начинающие не знают.
Именно осмысление подобного рода "деталей" и дает то, что называется профессиональной уверенностью.
Для Вашего удобства, сведу результаты в таблицу:

Таблица состояний битов Z и C.

- **нулевой результат**: бит **Z** = **1**,
- **ненулевой результат**: бит **Z** = **0**,
- **перенос есть**: бит **C** = **1**,

- **переноса нет**:	бит **C** = **0**,
- **заём есть**:	бит **C** = **0** (заем имеет инверсное значение).
- **заёма нет**:	бит **C** = **1** (заем имеет инверсное значение).

Перед тем, как работать с флагом **DC**, внесу некоторую ясность в вопрос о том, почему арифметические операции сложения и вычитания (в пределах одного байта) выполняются в ПИКах так быстро?
Или этот вопрос можно сформулировать так: "Почему флаги **C** и **DC** так быстро "реагируют" на результаты исполнения арифметических операций?"
Все познается в сравнении и поэтому приведу пример "доисторического" 4-хбитного арифметического устройства на двоичном счетчике с предустановкой 555ИЕ7 (полный цикл счета - 16 чисел).
Например, нужно сложить числа **.02** и **.10**.
Код числа **.02** подается на входы предустановки и записывается в триггеры счетчика.
После этого, на счетный вход счетчика подается 10 тактовых импульсов.
Счет начинается от числа **.02** и прекращается на числе **.12**.
Это - пример арифметического устройства последовательного типа, которое работает "долго и нудно".
И в самом деле, ждать результата 10 тактов - "радость небольшая", а если этот "динозавр", например, многокаскадный и речь идет о больших числах?
Такие вычисления могут понравиться только любителям перекуров.
Для того чтобы произвести эти вычисления всего за время одного такта, в микроконтроллерах используются так называемые 8-разрядные полные сумматоры.
Кто этим интересуется: информации о них полным-полно в технической литературе широкого пользования.
Почему именно "сумматор" - понятно, так как он осуществляет операцию суммирования.
А как быть с операцией вычитания, ведь сумматор может только складывать?
Дело в том, что применяя, при выполнении операции вычитания, специальные приемы (преобразования чисел), можно свести операцию вычитания к операции сложения.
В этом случае, операция вычитания является специфической разновидностью операции сложения, и с использованием полного сумматора, можно реализовать обе эти операции, что и имеет место быть.
Как конкретно это происходит, программисту знать вовсе не обязательно, но если кто интересуется, то литературы на этот счет много.
8- разрядный полный, универсальный сумматор работает в стандартном, параллельном, весовом (обычном) коде и имеет 2 группы входов, по 8 разрядов каждая, на которые "подаются" слагаемые числа, 8 выходов, на которые выводится результат суммирования и один выход переноса - заёма.
Для "запуска механизма" суммирования, в этом случае, никаких дополнительных сигналов управления не нужно, так как, образно выражаясь, полный сумматор суммирует, "не спрашивая разрешения".
Какие числа "выставлены" на входах, такие он тут же и суммирует.
Просто необходимо, в нужный момент, их "выставить", и сразу же после этого (буквально после паузы в несколько десятков наносекунд), можно считывать результат (он будет зафиксирован в течение всего интервала времени фиксации слагаемых чисел).
То есть, арифметические операции можно совершать очень быстро.
Таким образом, состояния бита флага **C** (**DC**) есть состояния выхода переноса - заёма такого универсального (задействуется и при сложении, и при вычитании), 8-разрядного полного сумматора.
Программист может только произвести опрос (программными средствами) состояний выхода переноса-заёма такого полного сумматора (что и есть опрос состояний флагов **C**/**DC**) и при необходимости, по результатам этого опроса, направить рабочую точку программы в один из двух сценариев.
Разрядность полного сумматора программист нарастить не может.
Он производит операции только в пределах одного байта.
После завершения операции, байт результата должен быть своевременно скопирован в регистр общего назначения, а иначе, в результате исполнения следующей операции, он будет "потерян" ("за что кровь пролита"?).
Продолжу тему флагов.

"Пытаю" флаг **DC**.
"Орудия пытки" → такие же, как и при "пытке" флага **C**, только контролировать нужно не нулевой бит регистра **STATUS**, а первый его бит.
В тексте программы **Multi**, замените проверочную группу команд для флага **C** на проверочную группу команд для флага **DC**.
Специфика работы с флагом **DC** такова, что лучше сначала прочитать пояснения, а потом начать с ним работать.
Байт состоит из 2-х полубайтов по 4 бита в каждом.
Они разделяются на **младший и старший полубайты**.
Младший полубайт - тот, в состав которого входит нулевой бит байта.
Так как в каждом полубайте по 4 бита, то младший полубайт способен отобразить 16 чисел (от **.00** до **.15**).
Можно провести аналогию между байтом, в случае работы с флагом **C**, и младшим полубайтом, в случае работы с флагом **DC**.
Всё один к одному. Разница только в том, что в первом случае, переносы или заёмы происходят в случаях "выхода" результата арифметической операции, за пределы числового диапазона байта (от **.00** до **.255**), а во втором случае, они происходят в случаях "выхода" результата арифметической операции, за пределы числового диапазона полубайта (от **.00** до **.15**).
При работе с флагом DC, перенос осуществляется в младший разряд старшего полубайта, а не в дополнительно задействованный регистр общего назначения, как в случае наращивания разрядности при работе с флагом C.
Если флаг **DC** работает с младшим полубайтом, то можно сделать следующий вывод:
флаги C и DC работают в комплексе.
Давайте посмотрим, как это будет выглядеть практически.
"Привяжемся" к операции суммирования.
Давайте что-нибудь посчитаем с использованием все той же группы проверочных команд для случая суммирования (см. выше).
Можете убрать из нее команду **bcf Status,0** по той причине, что при работе этой группы команд в составе программы **Multi.asm**, перед исполнением команды **addlw Y**, биты **C** и **DC** сброшены (флаги **C** и **DC** опущены) и эта команда (**bcf Status,0**) просто не нужна (это как раз то, о чем говорилось выше).
Оставшиеся 2 команды "вставьте" в текст программы **Multi.asm** на свое "проверочное место" (так, как это делалось ранее).
На место условных чисел **X** и **Y**, поставьте нули (начало анализа).
По ходу работы, можно производить и одновременную смену констант, но для того чтобы не запутаться, изначально, выгоднее зафиксировать одну из этих констант зафиксировать (**.00**).
Например, **X**, а константу **Y** постепенно увеличивать, начиная с числа **.00**.
Основное внимание нужно направить на нулевой и первый биты регистра **STATUS**.
Откройте окна **RAM** или **SFR**. Кому какое удобнее. А можно и оба.
Откройте также окно ***Watch***, "заложив" в него регистр **W** (в нем Вы будете видеть результат суммирования).
Не забывайте, после каждой смены значения константы, производить ассемблирование.
Напоминаю: используемая нами команда **ADDLW** воздействует на флаги **Z, C, DC** (см. распечатку команд).
"Встаньте" на команду **movlw .00**.
Это означает, что последней была выполнена команда **bcf Status,5**.
Посмотрите на содержимое регистра **STATUS**.
В нем - число **18h** (**00011000**).
Разбираемся: биты флагов **-PD** (3-й бит) и **-TO** (4-й бит) установлены в **1**.
Причина: после ассемблирования (что эквивалентно включению питания), ПИК сбрасывает сторожевой таймер **WDT**, то есть, автоматически исполняется команда **CLRWDT** (см. распечатку команд).
Почему при сбросе **WDT** 3-й и 4-й биты устанавливаются в **1**, а не в **0**?
Обратите внимание на **дефис** перед названием этих флагов: **он означает инверсию**.
Таким образом, флаги **-PD** и **-TO** опущены (установлены в **1**).
Далее, на эти флаги можно не обращать внимания.
Флаги **C** и **DC** также опущены (установлены в 0).
Производим суммирование двух нулей, последовательно выполняя команды **movlw .00** и

addlw .00.
После этого, содержимое регистра **STATUS** изменится с **18h** на **1Ch** (**00011100**).
Все правильно, **0+0=0**. Флаг нулевого результата **Z** (2-й бит) поднялся.
В команде **addlw**, меняем **.00** на **.01**.
Ассемблируем.
Сбрасываем программу на начало.
В пошаговом режиме доходим до команды **addlw .01** и исполняем ее.
Смотрим на содержимое регистра **STATUS**.
В нем опять число **18h** (**00011000**).
???
Вывод: так как результат суммирования не равен **0**, флаг нулевого результата **Z** сбросился.
Переносов нет, так как значение суммы не выходит "за границы (см. выше) рабочих зон".
Точно такая же "картина" будет наблюдаться, если Вы будете, в качестве числа **Y**, использовать любое число до **.15** включительно.
При этом, значение числа суммы не превысит **.15** и переполнения младшего полубайта не будет.
Суммируем **.00** и **.16**.
Сумма равна **.16**.
Это число большее, чем способен отобразить младший полубайт и казалось бы, флаг **DC** должен подняться).
Смотрим на содержимое регистра **STATUS**.
В нем **18h** (**00011000**). Почему флаг **DC** не поднялся?
Посмотрите на младший полубайт числа **.16**.
Он равен **.00** (**0000**), а **0+0=0**.
То есть, "выхода за границы" младшего полубайта нет, и флаг **DC**, по определению, подняться не должен.
Возникает вопрос: "Почему, когда мы раньше суммировали нули, флаг нулевого результата **Z** поднялся, а теперь он не поднялся"?
Ответ: флаг **Z** работает с байтом, а **.00**+**.16**=**.16**.
То есть, результат не является нулевым (в отличие от **.00**+**.00**=**.00**).
Точно так же флаг **DC** (а также флаги **C** и **Z**) не поднимется и в тех случаях, если Вы будете, вместо числа **Y**, "подставлять" любые другие числа, большие чем **.16** (вплоть до **.255**).
Можете поменять местами числа **X** и **Y**, результат будет тот же самый.
Учитывая то, что флаг **DC** не поднимался и тогда, когда вместо числа **Y** "подставлялись" числа от **.00** до **.15**, можно утверждать следующее: ***если одно из слагаемых (или оба) является нулем, то при любом числовом значении второго слагаемого, флаги C и DC никогда не поднимутся, а флаг Z поднимется только в случае .00+.00=.00.***
На основании этого утверждения, можно сделать следующий практический вывод: ***если одно из слагаемых равно нулю, то переносов происходить не будет.***
В сущности, это объясняется просто: в этом случае (одно слагаемое равно **0**), из-за того что другое слагаемое, по определению, не может быть больше числа **.255** (для байта) и **.15** (для полубайта), выхода, отображаемых байтом и полубайтом, чисел, за границы соответствующих "зон", не происходит.
Следовательно, не происходит и переносов. Поэтому флаги **C** и **DC** будут опущены.
А теперь замените **.00** на **.01**, а вместо **.16** поставьте **.15**.
Сумма - такая же, как и в случае **.00+.16=.16**. Посмотрим, что изменится.
После выполнения команды **addlw**, в регистре **STATUS** зафиксируется число **1Ah** (**00011010**).
Теперь флаг **DC** поднялся.
Он также поднимется и в случаях 2+14, 3+13, 4+12 и т.д. до 15+1 включительно, а также и при других комбинациях чисел, если ни одно из них не является нулем и их сумма больше **15**-ти.
А если меньше **15**-ти (в том числе и в случаях "применения" нулей), то флаг **DC** будет сброшен.
А теперь посмотрим, что произойдет, если слагаемые "выходят" из числового диапазона полубайта (**.00** … **.15**) в других случаях?
Например, просуммируем числа **.08** (**00001000**) и **.150** (**10010110**).
Флаг **DC** сбросился.
Почему?
Еще раз напоминаю, что флаг **DC** работает только в пределах младшего полубайта.

Младший полубайт числа **.08** равен **.08**. Младший полубайт числа **.150** равен **.06**.
Флаг **DC** сбросился потому, что **.08+.06=.14** (меньше **.15**-ти).
А теперь просуммируйте числа, например, **.08** (**00001000**) и **.153** (**10011001**).
Флаг **DC** поднялся.
Почему?
Младший полубайт числа **.08** равен **.08**. Младший полубайт числа **.153** равен **.09**.
Флаг **DC** поднялся потому, что **.08+.09=.17** (больше **.15**-ти).
А вот пример, когда оба флага (**C** и **DC**) поднимаются.
Просуммируйте числа, например, **.139** (**10001011**) и **.157** (**10011101**).
Флаги **C** и **DC** поднялись.
Почему?
Младший полубайт числа **.139** равен **.11**. Младший полубайт числа **.157** равен **.13**.
Флаг **DC** поднялся потому, что **.11+.13=.24** (больше **.15**-ти).
Флаг **C** поднялся потому, что **.139+.157=.296** (больше **.255**-ти).
После исполнения команды **addlw**, в регистр **W**, будет записано число **28h** (**.40**).
Разбираемся, откуда оно "взялось".
Из-за того, что число **.296** больше числа **.255**, произошел переход на второй "виток" счета.
В пределах этого второго "витка", результат суммирования "займет положение" числа **.296**-**.256**=**.40** (**28h**).
Количество этих "витков" не может быть более двух (почему? Ответьте самостоятельно).
Таким образом, вне зависимости от значения суммы двух чисел, отображаемой байтом, флаг **DC** "реагирует" только на результаты выполнения операций сложения (вычитания) содержимого младших полубайтов.
Образно выражаясь, флагу **DC** "все-равно, что происходит" в старшем полубайте.
Он работает на своем "узком участке" в 4 бита и "контролирует" именно этот участок, а не весь байт.
Весь байт (оба полубайта) "контролируют" флаги **Z** и **C**.
Возникает вопрос: "Где может дислоцироваться этот легендарный байт, с которым мы все время работаем?"
Ответ: там, куда программист посчитает целесообразным его поместить.
В данном случае, "местом его дислокации" является регистр **W** (см. комментарий к команде **ADDLW**).
Если же применить байт-ориентированную команду **ADDWF**, то можно выбрать, куда поместить результат операции: или во всё тот же аккумулятор, или в регистр общего назначения, а это уже "выход на широкий, оперативный простор".
А теперь посмотрите в комментарий к флагу **DC** (см. распечатку).
В нем флаг **DC** называется **флагом десятичного переноса - заёма**.
Этот комментарий я, в свое время, взял из технической документации, предоставляемой компанией Microchip.
Флаг **DC** не "привязан" к десятичной системе исчисления, так как для того чтобы осуществить такую "привязку", нужно "привязать" перенос-заём не к числу **.15**, а к числу **.9**.
Этого нет.
Флаг **DC** "привязан" только к двоичной системе исчисления (так же, как и остальные флаги).
Так что, слово "десятичного", из текста комментария к флагу **DC** лучше либо вообще убрать, либо заменить его на слово "полубайтного".
Флаг DC может применяться только для контроля типа "перенос - заем есть/нет" в "зоне" 4-хбитных чисел (**.00 … .15**).
Каким-то образом "аппаратно вычленить", из байта, младший полубайт и работать с ними отдельно, технически невозможно, так как все регистры ПИКов (по крайней мере, среднего семейства) 8-битные.
То, что "получается на выходе" - всегда 8-битное число.
Поэтому, при анализе "реакции" флага **DC** на результат соответствующей операции, приходится, из 8-битного результата, "вычленять" 4-хбитное число.
Только после этого можно ответить на вопрос: "Почему флаг **DC** поднялся или опустился?"
Если же производятся операции с 8-битными числами из числового диапазона от **.00 … .15**, то никаких проблем нет.
С учетом специфики флага **DC**, практическая надобность в его использовании, по сравнению с другими флагами, мала.

Если нужно "привязать" ход исполнения программы к событиям переносов/заёмов, то в подавляющем большинстве случаев, используется флаг **C**.
А теперь давайте остановимся, "переведем дух" и постараемся понять смысл прочитанного.
Может возникнуть вопрос: "Зачем нужно было так подробно расписывать этот въедливый процесс? Дал бы готовые определения, и дело с концом."
Ответ: при таком подходе к работе, программист рискует оказаться один на один со множеством неразрешимых задач, которые он не сможет решить даже при сильнейшем желании это сделать.
Таких задач, по ходу составления и отладки программы, может быть много, и если не владеть эффективной и агрессивной методикой их решения (типа "я тебе покажу кузькину мать!" или "не хочешь отдавать? Возьму силой"), то участь этого программиста печальна.
Какой смысл "взращивать" в себе комплекс неполноценности по отношению, например, к тем же флагам **C** и **DC**, если пользуясь достаточно простыми методиками, и имея в своем распоряжении такой мощнейший "инструмент" как **MPLAB**, можно "выпотрошить/взломать" не только эти флаги, но и многое другое.
Спокойно и по-деловому. Без обманчивых надежд на "манну небесную".
Польза такого рода "взломов" не столько в том, что "добывается" информация, сколько в том, что она как бы "становится родной" и поэтому легко "уходит в подкорку".
В виде опыта и индивидуальных, для каждого программиста, образов, которые, в нужное время, "извлекаются" из подсознания в виде готового (или почти готового) решения, которое не нужно каждый раз "вымучивать" и "рожать при помощи кесарева сечения".
Для того чтобы такое "извлечение" происходило в "автомате", нужно основательно "попотеть" и "провести в симуляторе" много времени, производя при этом какие-то осознанные, целенаправленные действия, с "уходом" в заблуждения и "выкарабкиванием" из них.
Примерно такого рода процесс я и попытался воспроизвести для затруднительной ситуации типа "как работают флаги **C** и **DC**?".
Это также можно рассматривать как примитивный пример программного анализа ("взлома").
Вывод: "не стоит ждать милостей от природы".
Если программист хочет добиться чего-то стОящего, то ему нужно не ждать "бесплатного сыра", а со знанием дела, "брать в руки интеллектуальную дубину и выбивать всю дурь из своих затруднений и проблем".
Нашему брату не нужно объяснять, как с ней обращаться.
Лишь бы она была в наличии, да потяжелее.
Плюс хорошие мозги.
Вот так и рождаются классные программисты.
Теперь, поближе к практике.
В следующем разделе, будет приведен пример использования флага **C**.

14. Пример задействования флага C в трехбайтном суммирующем устройстве. Циклический сдвиг. Операция умножения.

Пример реализации 3-байтного суммирующего устройства.

Составляя любую программу, программист работает с тем или иным количеством регистров общего назначения.
С их содержимым производятся различные операции.
Все они - однобайтные, и в пределах одного байта, работа с ними не вызывает особых затруднений.
Но часто возникает необходимость в работе с числами, значения которых велики (более, чем .255).
Например, мой частотомер содержит 8 десятичных разрядов, что соответствует максимальному значению отображаемого, десятичного числа **99 999 999**.
В двоичном виде, это число выглядит так: **101 11110101 11100000 11111111** (можете убедиться в этом в конверторе систем исчислений).
Итого, 27 двоичных разрядов.
Для того чтобы отобразить такое число, трех регистров окажется мало (8x3=24 разряда), следовательно, нужно задействовать 4 регистра общего назначения (8x4=32 разряда).
При этом получается "перебор" в 5 разрядов, но во-первых, "кашу маслом не испортишь" (эти 4 регистра способны отобразить число гораздо большее, чем 99999999), а во-вторых, выбора все-равно нет, так как все регистры 8-битные.
Вопрос: как распределить эти разряды по регистрам?
Ответ: необходимо определить порядок старшинства.
Вспоминайте о том, о чем говорилось ранее (HH, H, M, L).
Назначаем/"прописываем" регистры, например, с названиями **RegHH, RegH, RegM, RegL** и "закрепляем" за каждым из них по одному байту 4-байтного (32-битного) двоичного числа, начиная с младшего (он будет "лежать" в регистре **RegL**) и далее, в порядке возрастания старшинства, вплоть до самого старшего (он будет "лежать" в регистре **RegHH**).
Таким образом, мы распределили 4-байтное, двоичное число по 4-м регистрам общего назначения.
Таким образом, количество регистров, входящих в состав многобайтного регистра, назначается по принципу "перекрытия максимально возможного числа".
В данном примере, нужно 27 разрядов, а организовано 32.
Это единственный вариант с минимально возможным "перебором".
Например, в случае применения линейки из семи 7-сегментных индикаторов, достаточно и трех регистров (**RegH, RegM, RegL**).
А теперь давайте осмыслим, что мы "сотворили".
Мы "раздробили" единое, 4-байтное число, на 4 части, нарушив тем самым **внутренние связи, существующие между байтами**.
Проще говоря, имеются 4 отдельных части, которые, если не принять каких-то мер, между собой взаимодействовать не будут.
Следовательно, **нужно каким-то образом организовать это взаимодействие**.
Необходимость этой организации можно объяснить на таком примере.
Имеется какое-нибудь устройство на микросхеме 8-разрядного регистра (ИР).
Требуется собрать то же самое, но на 2-х микросхемах 4-разрядных регистров.
Следовательно, нужно соответствующим образом соединить эти две микросхемы, ведь без такого соединения, устройство не будет работать так, как нужно.
Примерно то же самое нужно сделать и в нашем случае.
Разница заключается в том, что в одном случае, речь идет об аппаратном способе соединения, а в другом случае, о программном способе соединения.
Ради этого и затеивались "разборки" с флагами.
То есть, взаимодействие байтов, "внутри" многобайтного числа, можно организовать с помощью флагов.
То же самое, но "с другого бока": с помощью флагов, из нескольких регистров общего назначения, можно создавать N-байтные регистры (N - количество задействованных регистров общего назначения).
Специфика организации работы таких регистров состоит в том, что многобайтный регистр "рождается" только тогда, когда программа этого "захочет".

То есть, в том случае, когда организовано программное взаимодействие между соответствующими байтами.
Если это взаимодействие организовано, то несколько регистров общего назначения будут работать как один многобайтный регистр.
На примере "учебно-тренировочного", суммирующего устройства, файлы которого называются **c_dc.asm** и **c_dc_1.asm** ("иллюстрация" принципа), давайте разберемся, как с помощью программных средств, организуется трехбайтный регистр, и как он работает?

Файл программы, с формализованными комментариями, называется c_dc.asm (находится в папке ***"Тексты программ"***).

Это выглядит так:

```
;*******************************************************************************
;          ПРИМЕР ИСПОЛЬЗОВАНИЯ ФЛАГА "C" (А ТАКЖЕ И ФЛАГА "Z")
;*******************************************************************************
; c_dc.asm    Стандартная подпрограмма суммирования двух трехбайтных чисел
;*******************************************************************************
            RegH, RegM, RegL - трехбайтное число 1-го слагаемого.
        TimerH, TimerM, TimerL - трехбайтное число 2-го слагаемого.
        Результат суммирования сохраняется в регистрах TimerH, TimerM, TimerL.
;*******************************************************************************
;                                  "ШАПКА ПРОГРАММЫ"
;===============================================================================
;.....................................................
;.....................................................
;===============================================================================
; Определение положения регистров специального назначения.
;===============================================================================
Status      equ         03h         ; Регистр Status.
;.....................................................
;.....................................................
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
RegL        equ         0Ch         ; Регистр младшего разряда 1-го слагаемого.
RegM        equ         0Dh         ; Регистр среднего разряда 1-го слагаемого.
RegH        equ         0Eh         ; Регистр старшего разряда 1-го слагаемого.

TimerL      equ         1Ch         ; Регистр младшего разряда 2-го слагаемого.
TimerM      equ         1Dh         ; Регистр среднего разряда 2-го слагаемого.
TimerH      equ         1Eh         ; Регистр старшего разряда 2-го слагаемого.
;.....................................................
;.....................................................
;===============================================================================
; Определение места размещения результатов операций.
;===============================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;.....................................................
;===============================================================================
; Присваивание битам названий.
;===============================================================================
C           equ         0           ; Бит флага переноса-заёма.
Z           equ         2           ; Бит флага нулевого результата.
;.....................................................
;===============================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*******************************************************************************


;*******************************************************************************
;                                  РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
```

```
;*******************************************************************************
START       .....................................
            .....................................
YES         .....................................
            .....................................
            goto    PLUS
            .....................................
            .....................................
;===============================================================================
; Суммирование двух трехбайтных двоичных чисел. Результат суммирования -
; трехбайтное двоичное число.
;===============================================================================
PLUS        movf    RegL,W          ; Скопировать содержимое регистра RegL
                                    ; в регистр W.
            addwf   TimerL,F        ; Сложить содержимое регистров W и TimerL с
                                    ; сохранением результата в регистре TimerL.
            btfss   Status,C        ; Опрос бита № 0 (флага C) регистра Status.
            goto    PLUS_1          ; Если бит № 0 =0, то переход в ПП PLUS_1.
            incf    TimerM,F        ; Если бит № 0 =1, то инкремент содержимого
                                    ; регистра TimerM с сохранением результата
                                    ; в нем же.
            btfss   Status,Z        ; Опрос бита № 2 (флага Z) регистра Status.
            goto    PLUS_1          ; Если бит № 2 =0, то переход в ПП PLUS_1.
            incf    TimerH,F        ; Если бит № 2 =1, то инкремент содержимого
                                    ; регистра TimerH с сохранением результата
                                    ; в нем же.
PLUS_1      movf    RegM,W          ; Скопировать содержимое регистра RegM
                                    ; в регистр W.
            addwf   TimerM,F        ; Сложить содержимое регистров W и TimerM с
                                    ; сохранением результата в регистре TimerM.
            btfsc   Status,C        ; Опрос бита № 0 (флага C) регистра Status.
            incf    TimerH,F        ; Если бит № 0 =1, то инкремент содержимого
                                    ; регистра TimerH с сохранением результата
                                    ; в нем же.
            movf    RegH,W          ; Если бит № 0 =0, то скопировать содержимое
                                    ; регистра RegH в регистр W.
            addwf   TimerH,F        ; Сложить содержимое регистров W и TimerH с
                                    ; сохранением результата в регистре TimerH.
            goto    YES             ; Безусловный переход в ПП YES.
;-------------------------------------------------------------------------------
            .....................................
            .....................................
;*******************************************************************************
            end                     ; Конец программы.
```

```
================================================================================
                АРИФМЕТИЧЕСКИЕ ПРИМЕРЫ СУММИРОВАНИЙ
================================================================================
                  Случай отсутствия переносов
--------------------------------------------------------------------------------
                       80   120   100
                       60    50   100
                      ----------------
                      140   170   200  Значения чисел сумм не превышают 255.
--------------------------------------------------------------------------------
                   Случай наличия переносов № 1
--------------------------------------------------------------------------------
                                        Младший разряд
                150          120          100
                100          180          200
                ------------------------------
                250+1=251    300+1-256=45 300-256=44
                ------------------------------
                251           45           44
```

```
При суммировании младших и средних разрядов значения сумм превысили 255
(2 переноса).
-------------------------------------------------------------------------------
    Случай наличия переносов № 2. Дополнительно задействуется флаг Z
                    (опрос его состояния).
-------------------------------------------------------------------------------
                                          Младший разряд
                  150           130           100
                  100           125           200
                  ------------------------------
                  250+1=251     255+1-256=0   300-256=44
                  --------------------------------
                  251             0            44
При суммировании младших и средних разрядов значения сумм превысили 255
(2 переноса).
-------------------------------------------------------------------------------
```

Файл программы, с неформализованными комментариями, называется c_dc_1.asm (находится в папке ***"Тексты программ"***).

Это выглядит так:

```
;*******************************************************************************
;          ПРИМЕР ИСПОЛЬЗОВАНИЯ ФЛАГА "C" (А ТАКЖЕ И ФЛАГА "Z")
;*******************************************************************************
; c_dc_1.asm   Стандартная подпрограмма суммирования двух трехбайтных чисел
;*******************************************************************************
            .....................................
            .....................................
            .....................................
            .....................................
;===============================================================================
; Суммирование двух трехбайтных двоичных чисел. Результат суммирования -
; трехбайтное двоичное число.
;===============================================================================
PLUS        movf    RegL,W          ; Скопировать содержимое регистра RegL
                                    ; в регистр W.
            addwf   TimerL,F        ; Сложить содержимое регистров W и TimerL с
                                    ; сохранением суммы в регистре TimerL.
            btfss   Status,C        ; Флаг C поднят?
            goto    PLUS_1          ; Если нет, то переход в ПП PLUS_1.
            incf    TimerM,F        ; Если да, то инкремент содержимого регистра
                                    ; TimerM с сохранением результата в нем же.
            btfss   Status,Z        ; Флаг Z поднят?
            goto    PLUS_1          ; Если нет, то переход в ПП PLUS_1.
            incf    TimerH,F        ; Если да, то инкремент содержимого регистра
                                    ; TimerH с сохранением результата в нем же.
PLUS_1      movf    RegM,W          ; Скопировать содержимое регистра RegM
                                    ; в регистр W.
            addwf   TimerM,F        ; Сложить содержимое регистров W и TimerM с
                                    ; сохранением суммы в регистре TimerM.
            btfsc   Status,C        ; Флаг C поднят?
            incf    TimerH,F        ; Если да, то инкремент содержимого регистра
                                    ; TimerH с сохранением результата в нем же.
            movf    RegH,W          ; Если нет, то скопировать содержимое
                                    ; регистра RegH в регистр W.
            addwf   TimerH,F        ; Сложить содержимое регистров W и TimerH с
                                    ; сохранением результата в регистре TimerH.
            goto    YES             ; Безусловный переход в ПП YES.
;-------------------------------------------------------------------------------
            .....................................
            .....................................
```

Проект под них создавать не нужно.

Просто откройте их в **MPLAB** (***File*** → **Open**).
Вы видите перед собой подпрограмму, которая реализует трехбайтное, суммирующее устройство.
В данном случае, осуществляется суммирование 3-хбайтного результата измерения частоты и 3-хбайтной поправки (константы), которая представляет собой значение установленной пользователем (или по умолчанию) промежуточной частоты (пример взят из программы частотомера-цифровой шкалы).
Вспомните первый случай вычисляемого перехода, который был рассмотрен ранее.
Одной из команд этого вычисляемого перехода является команда **goto PLUS**.
Таким образом, при наличии определенного состояния клавиатуры, происходит безусловный переход в ПП **PLUS**, в которой и осуществляется указанное выше суммирование.
Теперь нужно кое-что пояснить.
Микроконтроллер работает только с двоичными числами.
То, что в тексте программы, форма представления чисел может быть иной, это всего-лишь "элемент" удобства (спасибо разработчикам **MPLAB**).
В конечном итоге, все сведётся к той или иной разновидности двоичных чисел, с которыми и будет происходить работа.
Пример.
Предположим, что возникла необходимость в отображении символов десятичной системы исчисления.
Например, в линейке индикаторов или в жидкокристаллическом модуле.
Для того чтобы "провернуть это дельце", сначала нужно произвести соответствующие операции с классическими, двоичными числами, вплоть до получения двоичного числа результата.
После этого, нужно перевести классическое, двоичное число результата, в двоично-десятичную форму (разновидность двоичной формы чисел).
После этого, нужно осуществить перекодировку двоично-десятичных чисел под те индикаторы, которые применяются (еще одна разновидность двоичной формы чисел).
После этого, результат такого перекодирования можно вывести на индикацию.
В любом из этих случаев, происходит работа с двоичными числами.
Вернее, с теми или иными их разновидностями.
Как осуществляется перекодирование, было рассмотрено ранее (второй пример вычисляемого перехода), а как осуществляется перевод двоичных чисел в двоично-десятичные, будет рассмотрено позднее.
Сейчас же, я покажу, как формируется трехбайтное двоичное число, предназначенное для дальнейшего отображения в линейке из 7-ми 7-сегментных индикаторов частотомера - цифровой шкалы (работа в подрежиме суммирования результата измерения и значения промежуточной частоты).
Подпрограмма **PLUS_1** находится внутри подпрограммы **PLUS**.
Следовательно, если речь идет о ПП **PLUS**, то имеются ввиду все команды подпрограммы, включая и команды, входящие в состав ПП **PLUS_1**.
Вспомните то, о чем говорилось в разделе, посвященному работе с EEPROM памятью данных.
В ее ячейки, предварительно, можно записывать какие-нибудь числа, которые затем используются в работе программы.
В рассматриваемом случае, в 3 ячейки EEPROM памяти данных, с учетом порядка старшинства, записывается, предварительно вычисленное программистом, трехбайтное, двоичное число, которое задает значение промежуточной частоты.
После включения питания, числа из этих ячеек, с соблюдением порядка старшинства, копируются в регистры **RegH, RegM, RegL** (см. текст программы).
В регистрах **TimerH, TimerM, TimerL**, на момент начала исполнения ПП **PLUS**, находится трехбайтное, двоичное число результата измерения частоты.
В ПП **PLUS**, суммирование происходит побайтно, начиная с младших байтов слагаемых.
То есть, сначала производится суммирование содержимого регистров **RegL** и **TimerL**, затем **RegM** и **TimerM**, а затем **RegH** и **TimerH**.
После выполнения команд сложения, производится опрос состояния флага **C**, по результатам которого делается вывод о наличии или отсутствии необходимости переноса в более старший разряд (инкремента содержимого, следующего по разрядности, регистра **Timer**).
Критерий прост: превышение или нет, числовым значением суммы, "границы" числового

диапазона байта (числа **.255**).
Если это превышение есть, то флаг **C** поднимается, и по факту этого, далее, осуществляется инкремент (+1) содержимого следующего, по старшинству, регистра.
"С другого бока": если такой инкремент происходит, то это означает то, что осуществлен перенос в следующий, по старшинству, разряд трехбайтного регистра **TimerH/TimerM/TimerL**.
При наличии этого переноса, значение байта следующего, по старшинству, регистра **TimerM**(**H**), сначала увеличивается на 1, а затем, получившийся результат, суммируется с содержимым регистра **RegM**(**H**).
Если после суммирования, переноса нет (**C**=**0**), то инкремента содержимого следующего, по старшинству, регистра не происходит (команда инкремента обходится).
После исполнения команд сложения, их результаты сохраняются все в тех же регистрах **Timer** (с "привязкой" к соответствующим разрядам L, M, H).
Обращаю Ваше внимание на то, что инкремента содержимого младшего байта (переноса в младший байт), находящегося в регистре **TimerL**, никогда не происходит.
По той причине, что он младший и переносить в него нечего.
Таким образом, в данном случае, перенос может происходить только из младшего разряда L, в средний разряд M и из среднего разряда M, в старший разряд H.
Так как перенос из самого старшего разряда осуществлять некуда, то программист должен рассчитать количество разрядов многобайтного регистра таким образом, чтобы исключить переполнение регистра самого старшего разряда (того, в котором сохраняется результат суммирования).
То есть, в части касающейся суммирования чисел самого старшего разряда, арифметический результат суммирования не должен превышать **.255**.
В противном случае, произойдет "выход на второй виток счета" (неверный результат суммирования).
Например, **3-хбайтный** регистр сможет безошибочно отобразить числа
от **0** до 256x256x256-1=**16777215**.
Если нужно отобразить результат суммирования больший, чем 16777215, то нужно организовывать 4-байтный регистр (добавить регистры **RegHH** и **TimerHH**) и соответственно, добавить, в подпрограмму **PLUS**, команды, организующие суммирование содержимого регистров **RegHH** и **TimerHH**, а также и перенос из разряда **H** в разряд **HH**.
Снизу от директивы **end**, Вы видите "арифметические примеры суммирований".

Пример № 1: случай отсутствия переносов.
В этом случае, связи между байтами не работают (так как переносов нет), и просто осуществляется последовательные, побайтные суммирования, в пределах пары одноименных байтов (L+L, M+M, H+H) регистров **Reg** и **Timer**, начиная с младших байтов (**RegL**+**TimerL**=**...**).
Результаты анализов состояний битов **C** и **Z** будут нулевыми.
По этой причине, рабочая точка программы будет "обходить стороной" все команды инкрементов.

Пример № 2: случай наличия переносов в разряды M и H, при отсутствии нулевого результата операций суммирования.
Побайтное суммирование начинается с младших байтов (L) регистров **Reg** и **Timer**.
В результате этого суммирования, происходит "выход" на второе кольцо 8-битного, полного цикла чисел (переполнение), и флаг **C** поднимается ("сигнал" переноса в следующий разряд).
После опроса состояния флага **C**, рабочая точка программы уходит в сценарий "инкремент содержимого регистра **TimerM**", что и есть практическое осуществление переноса.
После этого, происходит суммирование байтов средних разрядов (M) регистров **Reg** и **Timer**.
В результате этого суммирования, также происходит "выход" на второе кольцо 8-битного, полного цикла чисел (с учетом предшествующего инкремента содержимого регистра **TimerM**), и флаг **C** снова поднимается.
После этого, рабочая точка программы уходит в сценарий "инкремент содержимого регистра **TimerH**", что также есть практическое осуществление переноса.
После этого, происходит суммирование байтов старших разрядов (H) регистров **Reg** и **Timer**.
В этом случае, переноса быть не должно (мотивация - см. выше).
Всё. Процесс суммирования завершен.

После этого (а также и в других случаях) исполняется команда **goto YES**, и рабочая точка программы "покидает" ПП **PLUS** для того чтобы далее "делать какие-нибудь полезные дела".
Если предположить, что ПП **PLUS** входит в состав программы частотомера-цифровой шкалы, то после выхода рабочей точки программы из ПП **PLUS**, происходит ее безусловный переход в подпрограммы преобразования двоичных чисел в двоично-десятичные, затем, в ПП кодирования, после чего, результат перекодировки выводится в порт, для отображения в линейке 7-сегментных индикаторов.

Пример № 3: случай наличия переносов в разряды M и H, при наличии нулевого результата операции суммирования байтов (слагаемые - не нулевые).
Его разница с примером № 2 заключается в том, что при суммировании чисел, "лежащих" в "средних" байтах (M) регистров **Reg** и **Timer**, "выход" на второе кольцо 8-битного полного цикла чисел происходит на число равное нулю.
В этом случае, поднимается флаг нулевого результата **Z**, и перенос в старший разряд (то есть, инкремент содержимого регистра **TimerH**) осуществляется после опроса состояния не флага **C**, а флага **Z**.
Сразу возникает вопрос: "А какова надобность задействования, для переноса, флага нулевого результата **Z**, ведь казалось бы, можно обойтись только флагом **C**"?
Существуют такое понятие, как "наихудший вариант", на который всегда (а не только в этом случае) нужно ориентироваться.
В данном случае, он выглядит так: в регистре "лежит" число **.255**, а из более младшего разряда, осуществлен **перенос**.
В этом случае, **.255+1(перенос)=256-256=.0** (выход на самое начало 2-го "витка").
То есть, должен быть осуществлен инкремент содержимого следующего, по старшинству, байта.
Давайте разбираться.
Для данной программы, при суммировании разрядов L (**addwf TimerL,F**), флаг **C** работает, так как применяется команда (**addwf**), влияющая на его состояние (см. распечатку команд).
По этой причине, перенос в разряд M (при его наличии) будет безошибочно осуществлен.
А вот при организации переноса, из разряда M, в разряд H, в рассматриваемом случае, анализ состояния флага **C** просто бесполезен, так как флаг **C** не "реагирует" на команду **incf** (см. распечатку команд).
То есть, после инкремента числа **.255**, флаг **C** как был опущенным, так он таким и останется.
Соответственно, инкремента содержимого регистра **TimerH** произведено не будет, что есть ошибка суммирования.
Значит нужно "искать заменитель".
На числовой результат исполнения команды **incf** реагирует только флаг **Z** (см. распечатку команд), и состояние этого флага обязательно нужно опросить.
Соответственно, при организации переноса, из разряда M, в разряд H, для "нейтрализации вышеобозначенной бяки", организована группа команд анализа состояния флага **Z**.
Общий принцип: ***при наличии переноса, к содержимому регистра следующего, по старшинству разряда, тем или иным способом, нужно прибавить единицу.***
При отсутствии переноса, единица не прибавляется.
"Наихудший вариант" выглядит так (один из примеров):

```
150  255  255
0      0    1
151    0    0
```

С суммированием младших байтов (L), нет никаких проблем:
.255+.01=.256=.00 плюс перенос (инкремент) в регистр разряда M.
После инкремента содержимого регистра разряда M, флаг **Z** поднимется, после чего содержимое регистра разряда H будет увеличено на единицу: **.150+.0+.1переноса=.151**
Этот способ организации переносов, внутри многобайтного регистра, и реализован в предлагаемой Вашему вниманию ПП **PLUS**.
Предоставляется прекрасный повод для тренировки в составлении блок-схемы подпрограммы, и я советую Вам это сделать.
В дальнейшем, проще будет "ориентироваться".
ПП **PLUS** является типичным образцом подпрограммы с достаточно "развитой" системой

ветвлений. Кто ее "осилит", тот существенно "продвинется вперед".
Эту ПП ("заготовку"), в дальнейшем, можно использовать при разработке программ, в которых нужно производить суммирования чисел.
С непривычки, алгоритм этой подпрограммы может показаться достаточно сложными, но если разобраться, то ничего особо сложного в нем нет.
Нужно просто некоторое время "повариться в этом соку".
Именно по этой причине, а также по причине того, что стандартная подпрограмма вычитания сложнее стандартной подпрограммы сложения, я пока не считаю целесообразным производить "разборки" с подпрограммой вычитания.
Перед "штурмом этой вершины", нужно основательно разобраться с суммированием, причем, разобраться так, чтобы неясных вопросов не было.
Это обусловлено тем, что вычитание является специфической формой суммирования и программист, хорошо представляющий себе процесс суммирования многобайтных двоичных чисел, при "разборках" с вычитанием, не встретит особых трудностей.
Так что, всему свое время. Дойдет дело и до вычитания.
Теперь можно разобраться с парой очень востребованных операций, имеющих непосредственное отношение к флагу **C**, но сначала подведем промежуточный итог.
Итак, рабочие части текстов "неполноценных" программ **addwfpc.asm**, **c_dc.asm**, **flag.asm** (и им подобных), можно "врезать" в какую-нибудь простенькую, "полноценную" программу и от души "погонять все это добро" в симуляторе, выяснив при этом все, что нужно.
Естественно, что при этом необходимо "скорректировать шапку" программы и создать необходимые условия для работы этих "врезок".
Лучше всего, специально под это дело, создать, например, из программы **Multi.asm**, программу-заготовку, в "шапке" которой "оптом прописать" наиболее востребованные регистры специального назначения, определить место сохранения результатов операций и присвоить названия наиболее востребованным битам регистров специального назначения.
Например, **C equ 0**, **DC equ 1**, **Z equ 2**, **RP0 equ 5**, **GIE equ 7** и т.д.
Даже если часть из этого использоваться не будет, то ничего страшного.
Это не является ошибкой.
Примечание: если воспользоваться соответствующей директивой, которая "оптом прописывает всё, что шевелится", можно обойтись без этого. Эта директива комфортна, но на мой взгляд, изначально вредна. Так как расхолаживает. Это вовсе не говорит о том, что ее вообще не нужно применять. Нужно. Но позднее. Когда произойдет "вживание/вживление в шапку".
Итак, после такой корректировки, нужно только "прописать" использующиеся регистры общего назначения, после чего можно переходить к рабочей части программы.
В ней нужно создать условия для работы "врезки" (если это нужно), после чего с ней можно работать (выяснять интересующие Вас детали и/или видоизменять под свои задумки).
И в этом случае, и во многих других случаях, речь идет о "раскрутках логических цепочек", начиная с первых ее "звеньев" и далее, по порядку.
В ходе этих "раскруток", можно получить качественные ответы на бОльшую часть вопросов, связанные с работой как составных частей программы, так и всей программы.
Кроме того, такая работа развивает аналитические способности, а по своей обучающей эффективности ей вообще нет равных.
Эти "страшилки" я рассказываю Вам вовсе не для того чтобы Вы испугались объема предстоящей работы (если есть "упёртость", то это не помеха, а если ее нет, то тогда выгоднее испугаться), а для того чтобы чётче обозначить наиболее эффективное ее направление.
Эта работа не для лентяев (пусть даже самых одаренных), так как, особенно на первых порах, она связана с достаточно большими трудозатратами и повышенным количеством неудач.
А как же иначе?
Дети рождаются в муках, но и радости имеются, ведь самостоятельно составленная программа - своеобразный "ребенок".
Оччень дорогой, любимый и приносящий много радости.
"Родами" займемся позднее, а пока продолжим "курс молодого бойца".

Циклический сдвиг

Циклическим сдвигом "рулят" две команды: **RLF** (циклический сдвиг влево) и **RRF** (циклический сдвиг вправо).
Обе эти команды → байт-ориентированные.
Это означает то, что циклический сдвиг происходит внутри байта.
Посмотрите в распечатку команд.
В ней, в строках команд **RLF** и **RRF**, Вы найдете блок-схему циклического сдвига влево и вправо.
Механизм сдвига - кольцевой.
Старший или младший бит байта (в зависимости от направления сдвига) как бы "изымается" и "переносится", через бит флага переноса-заёма **C**, на противоположный "конец" байта.
В данном случае, работа флага **C** сводится к работе простейшего, однобитного (не путайте с однобайтным) регистра, в который, после исполнении команды циклического сдвига, копируется значение старшего или младшего бита (в зависимости от направления сдвига) байта регистра, к которому обращается команда **RLF** или **RRF**.
А можно и "зайти с другого конца", как бы введя бит **C** в состав регистра, к которому обращается команда **RLF** или **RRF**, с присвоением ему "статуса" бита № 8.
В этом случае, речь идет о составном (комплексном) 9-битном регистре, с раздельным отображением байта (биты с №№ 0 … 7) и бита с № 8, замкнутом в кольцо обратной связи.
Числовое значение байта можно проконтролировать, просмотрев содержимое регистра, к которому обращается команда **RLF** или **RRF**, а состояние бита с № 8 можно проконтролировать, просмотрев содержимое флага **C**.
В большинстве случаев, перед началом процедуры циклического сдвига, бит **C** нужно сбросить в ноль.
Если до начала этой процедуры, не применялись команды, воздействующие на флаг **C** (**ADDWF**, **ADDLW**, **SUBWF**, **SUBLW**), то о сбросе флага **C** можно не заботиться: он будет сброшен по умолчанию.
Если эти команды применялись, то на момент начала процедуры циклического сдвига, нужно проконтролировать состояние бита флага **C**, и если в нем установлена **1**, то нужно сбросить бит **C** в **0** командой **bcf Status,0**(**C**).
Если лень "забивать себе голову" выяснением состояния бита **C** на момент начала операции циклического сдвига, то непосредственно перед проведением этой операции, нужно просто сбросить бит **C** в **0**, и дело с концом.
Для того чтобы понять, как происходит процедура циклического сдвига, достаточно провести аналогию с работой реверсивного, сдвигового регистра с предустановкой типа ....ИР.. .
Хотя микросхем 9-разрядных регистров ИР и нет, но, предположим, что такой регистр существует.
Сейчас я Вам "обрисую" аналог процедуры циклического сдвига.
"Закольцовываем" регистр, соединяя выход 9-го разряда со счетным входом 1-го разряда.
8-битное исполнительное устройство, назовем его **A**, подключаем к выходам с 1-го по 8-й, а 1-битное исполнительное устройство, назовем его **B**, подключаем к 9-му выходу регистра.
Сначала, на 8-ми параллельных входах предустановки (начало нумерации - от младшего разряда) выставляются уровни, соответствующие числу, которое нужно преобразовать в результате сдвига (сдвигов), а на 9-м входе предустановки выставляем **0**.
Записываем все это в регистр.
Это соответствует записи, в регистр, числа, которое нужно преобразовать в результате исполнения процедуры циклического сдвига и предварительному сбросу бита флага **C**.
Далее, на вход управления реверсом, подается уровень, соответствующий, например, сдвигу влево.
Это соответствует применению команды **RLF**.
На тактовый вход регистра подается один тактовый импульс.
Это соответствует исполнению команды **RLF**.
Происходит сдвиг, предварительно записанной информации, в сторону старших разрядов (влево).
Из-за наличия "закольцовки", бит с **№ 8** "занимает место" бита с **№ 0**, а бит с **№ 7** "занимает место" бита с **№ 8** (остальные сдвигаются соответственно).
На управляющие входы исполнительных устройств **A** и **B** подаются уровни, сдвинутые на такт влево.
Таких тактовых импульсов (команд **RLF**) можно сформировать (исполнить) несколько, и

соответственно, содержимое 9-разрядного регистра сдвинется влево на такое же количество разрядов.
На 10-м сдвиге, в регистре установится исходное число.
А теперь поближе к ПИКам.
В одном регистре общего назначения, можно отобразить только 8 битов этого "виртуального" (условно), 9-битного числа (начиная с младшего), а 9-й (старший) бит находится как бы "за кулисами".
А теперь давайте проанализируем "механику этого действа" и какую практическую выгоду можно из этого извлечь.
"Прогоним" по полному циклу сдвигов влево, например, число **.09** (**00001001**).

```
00001001   C=0       9                  Начальная установка.
00010010   C=0  9x2= 18           x2
00100100   C=0  18x2=36           x4
01001000   C=0  36x2=72           x8
10010000   C=0  72x2=144          x16
00100000   C=1  144x2=288-256=32       Выход на 2-й "виток" цикла счета.
01000001   C=0  32x2+1{перенос, из бита C, в младший разряд байта}=65
10000010   C=0  65x2=130
00000100   C=1  130x2=260-256=4.       Выход на 2-й "виток" цикла счета.
.................................
00001001   C=0  4x2+1{перенос, из бита C, в младший разряд байта}=9
.................................
```

Первое, что бросается в глаза, это движение единиц.
"Привязавшись" к ним (или к нулям), можно реализовать, например, устройства типа бегущих огней, бегущей строки или что-то подобное.
А нельзя ли, при помощи циклических сдвигов, реализовать что-нибудь "посолиднее"?
Очень даже можно.
После анализа результатов циклического сдвига влево числа **.09** (или других чисел), можно сделать следующие выводы:

- ***Циклический сдвиг влево, на 1 такт*** (позицию)***, связан с операцией умножения "предшествующего"*** ("лежащего" в регистре до этого сдвига) ***числа на 2.***
- ***Если результат умножения на 2 "выходит на второй виток" полного цикла чисел*** (более **.255**)***, то флаг C поднимается*** (перенос)***.***
- ***Если в бите C находится 1, то результат следующего циклического сдвига, с учетом умножения на 2, увеличивается на 1.***

В случае использования однобайтного регистра, "закольцованного", через бит **C**, на самого себя (рассмотренный выше случай), последовательно исполняя несколько команд **RLF**, можно умножить исходное число на **2, 4, 8 ...** , но только при условии отсутствия переносов (см. приведенный выше пример).
Например, после умножения на **2** (один сдвиг) чисел до **.127** включительно, в регистре общего назначения, к которому обращается команда **RLF**, будет "лежать" число, точно отображающее результат выполнения арифметической операции умножения на **2**, а бит флага **C** всегда будет нулевым (старший бит байта всегда будет равен нулю).
После умножения на **2** чисел от **.128** до **.255** включительно, в регистре общего назначения, к которому обращается команда **RLF**, будет "лежать" число, отображающее результат выполнения арифметической операции умножения на **2**, но только с учетом "выхода на второй виток" цикла счета, а бит флага **C** установится в **1** (перенос).
В этом случае, для получения истинного результата умножения на **2**, необходимо к числу, "лежащему" (после исполнения команды **RLF**) в регистре общего назначения, к которому обращается команда **RLF**, прибавить число **.256**.
Но не будет же программист, по ходу исполнения программы, "вручную" вносить поправки в результаты вычислений. Нужно сделать так, чтобы он этим не занимался.
Вопрос: "Как"?
Ответ: нужно нарастить разрядность, задействовав один или несколько дополнительных, предварительно (до начала процедуры сдвигов) "обнуленных" регистров общего назначения.
В этом случае, исходное, однобайтное число можно умножить на множитель кратный двум

(1 сдвиг – **x2**, 2 сдвига – **x4**, 3 сдвига – **x8** и т.д.), причем, эта кратность может быть достаточно большой (зависит от количества задействованных регистров).
После окончания сдвигов, результат кратного умножения на **2** считывается из N-байтного регистра.
Такой способ умножения, хотя и прост, но если использовать только его, то его возможности сильно ограничены, да и с множителем равным двум в степени **N** (где **N** – количество сдвигов) "особо-то и не развернешься".
Поэтому, нужен комплексный подход.
А как, например, умножить на 7 или на 10 и т.д.?
В этом случае, используется несколько последовательных умножений на 2 (сдвигов влево), с учетом остатка.
Для этого нужно составить разложение числа (множителя).
Арифметика этого разложения простая.
Например, приемлемые варианты для **7**: 7=2x2+1+1+1 или 7=2x2x2-1.
В первом случае, посредством двух сдвигов влево, множимое умножается на 4.
Далее, к этому результату, 3 раза, последовательно прибавляется множимое.
Во втором случае, посредством трех сдвигов влево, множимое умножается на 8.
Далее, из этого результата, один раз вычитается множимое.
Какой вариант выбрать?
Лучше выбрать второй вариант, так как в этом случае, группа команд умножения будет содержать меньшее количество команд, чем в случае использования первого варианта.
При умножении на **10** могут быть такие варианты: 2x2x2+1+1, 2x2x2x2-1-1-1-1-1-1, (2x2+1)+(2x2+1), 2x2+1+1+1+1+1+1 и т.д.
Самым оптимальным является 1-й вариант, так как для его реализации, требуется наименьшее количество команд.
Вот его-то, в качестве примера, давайте и реализуем (простейшая ПП умножения).

Файл программы умножения на 10, с формализованными комментариями, называется Rlf.asm (находится в папке ***"Тексты программ"***).

Она выглядит так:

```
;*******************************************************************************
;      ПРИМЕР ИСПОЛЬЗОВАНИЯ ФЛАГА "C" В ОПЕРАЦИИ УМНОЖЕНИЯ НА 10
;*******************************************************************************
; Rlf.asm            Подпрограмма умножения 8-битных чисел на 10
;*******************************************************************************
; Множимое: числа от .00 до .255, которые закладываются в регистр RegL на момент
; начала подпрограммы перемножения на 10.
; Множитель: 10 (2*2*2+1+1).
; Произведение: результат трех умножений на 2 и двух суммирований. На момент
; окончания ПП умножения на 10, записывается в двухбайтный регистр RegL/RegH
; (двухбайтное двоичное число).
;*******************************************************************************
;                                  "ШАПКА ПРОГРАММЫ"
;*******************************************************************************
;.....................................................
;.....................................................
;==============================================================================
; Определение положения регистров специального назначения.
;==============================================================================
Status      equ         03h         ; Регистр Status.
;.....................................................
;.....................................................
;==============================================================================
; Определение названия и положения регистров общего назначения.
;==============================================================================
RegL        equ         0Ch         ; Регистр множимого.
                                    ; Он же - регистр младшего байта.
RegH        equ         0Eh         ; Регистр старшего байта.
;.....................................................
;.....................................................
```

```
;=============================================================================
; Определение места размещения результатов операций.
;=============================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;.............................................................
;=============================================================================
; Присваивание битам названий.
;=============================================================================
C           equ         0           ; Бит флага переноса-заёма.
;.............................................................
;=============================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*****************************************************************************


;*****************************************************************************
;                                РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;*****************************************************************************
START       .....................................
            .....................................
            .....................................
            .....................................
На момент начала ПП умножения, в регистр RegL должно быть записано однобайтное
двоичное число (множимое), которое нужно умножить на 10 (множитель).
Значения произведений отражаются в двухбайтном регистре RegL/RegH.
;=============================================================================
; Подпрограмма умножения на 10 однобайтных двоичных чисел (.00 ... .255)
;=============================================================================
; Подготовительные операции.
;-----------------------------------------------------------------------------
            bcf         Status,C    ; Установка в 0 бита № 0 регистра Status.
            movf        RegL,W      ; Копирование содержимого регистра RegL
                                    ; в регистр W.
            clrf        RegH        ; Сброс в 0 содержимого регистра RegH.
;-----------------------------------------------------------------------------
; Три последовательных умножения на 2 (умножение на 8).
;-----------------------------------------------------------------------------
            rlf         RegL,F      ; Циклический сдвиг влево содержимого
                                    ; регистра RegL с сохранением результата
                                    ; в нем же.
            btfsc       Status,C    ; Опрос состояния бита флага C.
            rlf         RegH,F      ; Если C=1, то циклический сдвиг влево
                                    ; содержимого регистра RegH с переносом в его
                                    ; младший разряд 1 из бита флага C, с
                                    ; сохранением результата в нем же (в RegH).
;-----------------------------------------------------------------------------
            rlf         RegL,F      ; Если C=0 (а также после исполнения
                                    ; предшествующей команды), то циклический
                                    ; сдвиг влево содержимого регистра RegL с
                                    ; сохранением результата в нем же.
            btfsc       Status,C    ; Опрос состояния бита флага C.
            rlf         RegH,F      ; Если C=1, то циклический сдвиг влево
                                    ; содержимого регистра RegH с переносом в его
                                    ; младший разряд 1 из бита флага C, с
                                    ; сохранением результата в нем же (в RegH).
;-----------------------------------------------------------------------------
            rlf         RegL,F      ; Если C=0 (а также после исполнения
                                    ; предшествующей команды), то циклический
                                    ; сдвиг влево содержимого регистра RegL, с
                                    ; сохранением результата в нем же.
            btfsc       Status,C    ; Опрос состояния бита флага C.
            rlf         RegH,F      ; Если C=1, то циклический сдвиг влево
                                    ; содержимого регистра RegH с переносом в его
                                    ; младший разряд 1 из бита флага C, с
```

```
                                    ; сохранением результата в нем же (в RegH).
                                    ; Если C=0 (а также после исполнения
                                    ; предшествующей команды), то программа
                                    ; исполняется далее (переход к суммированиям)
;-------------------------------------------------------------------------------
; 2 последовательных суммирования сохраненного в регистре W множимого, к
; результату умножения множимого на 8.
;-------------------------------------------------------------------------------
            addwf       RegL,F      ; Сложить содержимое регистров RegL и W с
                                    ; сохранением результата в регистре RegL.
            btfsc       Status,C    ; Опрос состояния бита флага C.
            incf        RegH,F      ; Если бит C=1, то инкремент содержимого
                                    ; регистра RegH с сохранением результата
                                    ; в нем же.
            addwf       RegL,F      ; Если бит C=0 (а также после исполнения
                                    ; предшествующей команды), то сложить
                                    ; содержимое регистров RegL и W, с
                                    ; сохранением результата в регистре RegL.
            btfsc       Status,C    ; Опрос состояния бита флага C.
            incf        RegH,F      ; Если бит C=1, то инкремент содержимого
                                    ; регистра RegH, с сохранением результата
                                    ; в нем же.
;-------------------------------------------------------------------------------
            ........................; Если бит C=0 (а также после исполнения
            ........................; предшествующей команды), то программа
                                    ; исполняется далее.
            ....................................
            ....................................
;*******************************************************************************
            end                     ; Конец программы.
```

Примечание: в конце ПП умножения, вместо сценария "программа исполняется далее", можно использовать сценарий условного или безусловного перехода в подпрограмму, находящуюся в любом "месте" текста рабочей части программы.
В этом случае, после команды последнего инкремента, должна быть исполнена команда `call` или `goto` соответственно.

Файл программы умножения на 10, с неформализованными комментариями, называется Rlf_1.asm (находится в папке ***"Тексты программ"***).

Она выглядит так:

```
;*******************************************************************************
;      ПРИМЕР ИСПОЛЬЗОВАНИЯ ФЛАГА "C" В ОПЕРАЦИИ УМНОЖЕНИЯ НА 10
;*******************************************************************************
; Rlf_1.asm          Подпрограмма умножения 8-битных чисел на 10
;*******************************************************************************
            ....................................
            ....................................
            ....................................
;===============================================================================
; Подпрограмма умножения на 10 однобайтных двоичных чисел (.00 ... .255)
;===============================================================================
; Подготовительные операции.
;-------------------------------------------------------------------------------
            bcf         Status,C    ; Предварительный сброс флага C.
            movf        RegL,W      ; Копирование содержимого регистра RegL
                                    ; в регистр W.
            clrf        RegH        ; Предварительный сброс содержимого
                                    ; регистра RegH.
;-------------------------------------------------------------------------------
; Три последовательных умножения на 2 (умножение на 8).
;-------------------------------------------------------------------------------
            rlf         RegL,F      ; Умножение на 2 числа, записанного в
                                    ; регистре RegL, с сохранением результата
```

```
                                    ; в нем же.
            btfsc       Status,C    ; Флаг C поднят или опущен ?
            rlf         RegH,F      ; Если поднят, то циклический сдвиг влево
                                    ; содержимого регистра RegH с переносом в его
                                    ; младший разряд 1 из бита флага C, с
                                    ; сохранением результата в нем же (в RegH).
;-------------------------------------------------------------------------------
            rlf         RegL,F      ; Если опущен (а также после исполнения
                                    ; предшествующей команды), то умножение на 2
                                    ; числа, записанного в регистре RegL, с
                                    ; сохранением результата в нем же.
            btfsc       Status,C    ; Флаг C поднят или опущен ?
            rlf         RegH,F      ; Если поднят, то циклический сдвиг влево
                                    ; содержимого регистра RegH, с переносом в
                                    ; его младший разряд 1 из бита флага C, с
                                    ; сохранением результата в нем же (в RegH).
;-------------------------------------------------------------------------------
            rlf         RegL,F      ; Если опущен (а также после исполнения
                                    ; предшествующей команды), то умножение на 2
                                    ; числа, записанного в регистре RegL, с
                                    ; сохранением результата в нем же.
            btfsc       Status,C    ; Флаг C поднят или опущен ?
            rlf         RegH,F      ; Если поднят, то циклический сдвиг влево
                                    ; содержимого регистра RegH, с переносом в
                                    ; его младший разряд 1 из бита флага C, с
                                    ; сохранением результата в нем же (в RegH).
                                    ; Если опущен (а также после исполнения
                                    ; предшествующей команды), то программа
                                    ; исполняется далее (переход к суммированиям)
;-------------------------------------------------------------------------------
; 2 последовательных суммирования сохраненного в регистре W множимого к
; результату умножения множимого на 8.
;-------------------------------------------------------------------------------
            addwf       RegL,F      ; Сложить результат умножения на 8 и множимое
                                    ; (сохраненное в регистре W), с сохранением
                                    ; результата в регистре RegL.
            btfsc       Status,C    ; Флаг C поднят или опущен?
            incf        RegH,F      ; Если флаг C поднят, то инкремент
                                    ; содержимого регистра RegH, с сохранением
                                    ; результата в нем же.
            addwf       RegL,F      ; Если флаг C опущен (а также после
                                    ; исполнения предшествующей команды), то
                                    ; сложить результат предшествующего
                                    ; суммирования и множимое, с сохранением
                                    ; результата в регистре RegL.
            btfsc       Status,C    ; Флаг C поднят или опущен?
            incf        RegH,F      ; Если флаг C поднят, то инкремент
                                    ; содержимого регистра RegH, с сохранением
                                    ; результата в нем же.
;-------------------------------------------------------------------------------
            ........................; Если флаг C опущен (а также после
            ........................; исполнения предшествующей команды), то
                                    ; программа исполняется далее.
            ....................................
            ....................................
;*******************************************************************************
            end                     ; Конец программы.
```

В свое время, когда я разбирался с умножением, мне нужно было выбрать: или найти готовую ПП умножения, или составить ее самому.
Я выбрал последнее. Так "родилось сие творение".
И дело даже не в том, что наиболее вероятным результатом подобного рода работы является "изобретение колеса", а в том что после нее, наступает сильнейшее "просветление", и приобретается ценный опыт.
В конечном итоге, такой "въедливый" и достаточно трудоемкий (на первых порах) стиль

"разборок с непонятностями", оказывается гораздо эффективнее, чем достаточно "мутный и ограниченный" стиль работы, связанный с поиском чего-то готового.
При этом, частенько, программист не вдается в детали того, что он находит.
По этой причине, велика вероятность возникновения затруднительных ситуаций.
Вплоть до "тупиков".
На мой взгляд, критерием успешности является что-то типа этого: "Зачем искать готовую, нужную программу/подпрограмму? Проще, быстрее и качественнее составить ее самому."
Итак, предположим, что нужно составить подпрограмму умножения.
Наиболее сложный "участок уже пройден".
Простейший принцип умножения сформулирован.
Ну и в чем дело? Как говорится, "вперед и с песней"!
Конструируем **подпрограмму умножения однобайтного двоичного числа на 10**.
Определяемся с регистрами общего назначения.
Так как множимое является числом, отображаемым одним байтом (**.00** ... **.255**), то такое множимое может быть записано в один регистр общего назначения.
Так как множимое может принять любое значение из диапазона чисел от **.00** до **.255**, то в результате циклических сдвигов, может происходить событие **переноса**.
Таким образом, максимальное значение произведения не может превысить числа

255x10=**.2550**

То есть, для отображения такого числа, одного регистра общего назначения мало и необходимо назначить еще один регистр общего назначения.
И в самом деле, двухбайтный регистр способен отобразить число от **0** до 256x256=65536-1=**.65535**, и число **.2550** попадает в этот числовой диапазон.
Так как регистров общего назначения стало 2, то нужно определиться с порядком старшинства.
Под младший байт произведения (он же - байт множимого), "прописываем" регистр общего назначения с названием, например, **RegL**.
В этом байте будет осуществляться циклический сдвиг влево, и он же будет младшим байтом произведения.
Под старший байт произведения, "прописываем" регистр общего назначения с названием, например, **RegH**.
В этот байт, в ходе последовательных процедур умножения на 2, будут осуществляться переносы.
Теперь нужно определиться с начальными условиями.
Для этого нужно ответить на вопрос: "Какие числа должны быть записаны в регистры **RegL, RegH** и в бит флага **C** на момент начала исполнения первого, циклического сдвига?"
Ответ: в регистр **RegL** будет записываться число множимого.
Байт регистра **RegH** нужно сбросить в **0** (**clrf RegH**).
В противном случае, это равносильно наличию переносов, которых не было, что неприемлемо (увеличенное, по сравнению с истинным, значение произведения).
Бит флага **C** нужно сбросить в **0** (**bcf Status,C**).
В противном случае, результат умножения будет не верен.
Посмотрите в текст программы.
В "подготовительных операциях", Вы увидите обе этих команды.
В "шапке" программы, Вы увидите, что регистры **RegL, RegH**, а также и регистр **Status**, "введены в эксплуатацию" ("прописаны").
Исходя из сформулированных выше условий, ПП умножения должна содержать не только группы команд умножения на 2, но и группы команд суммирования, причем, одним из слагаемых обязательно должно быть множимое, и операции сложения должны исполняться после завершения операции умножения на 8 (3-х операций умножения на 2).
Следовательно, в "подготовительных операциях" нужно сохранить множимое в каком-то регистре.
Под это дело, можно "прописать" еще один (третий) регистр общего назначения, а можно и скопировать множимое в регистр **W**, так как он, при исполнении процедур умножения на 2, не задействуется.
Поэтому, необходимости в увеличении количества задействованных регистров общего назначения, нет.
Вполне можно обойтись и двумя.
Следовательно, с целью дальнейшего использования множимого в группах команд

суммирования, в "подготовительных операциях", с помощью команды **movf RegL,W**, нужно сохранить множимое в регистре **W**.
Итак, группа команд "подготовительных операций" сформирована.
Порядок их расположения, в этой группе, не имеет значения.
Можно "перетасовать их, как карты". Это не отразится на результате работы подпрограммы, так как все 3 команды "развязаны" относительно друг друга.
Теперь можно заняться "конструированием" группы команд умножения на 2.
Возникает вопрос: "Какие именно команды должны входить в группу команд умножения на 2 и в каком порядке, в пределах этой группы, они должны располагаться?"
Исходя из сказанного ранее, однозначным является то, что первой должна следовать команда циклического сдвига влево **rlf RegL,F**, а второй, команда опроса бита флага **C** (**btfsc Status,C**), а также и то, что перенос, в случае его наличия, должен осуществляться в младший разряд старшего байта (в регистр **RegH**).
Что дальше? Каким образом должно происходить "взаимодействие" младшего и старшего байтов при переносах?
На момент начала первого циклического сдвига влево, в регистре **RegH** установлен **0**, и в дальнейшем, от переноса к переносу, значения чисел, "лежащих" в регистре **RegH**, должны только увеличиваться.
Вопрос: "По какому закону?"
Могут быть следующие варианты: суммирование, инкремент, циклический сдвиг влево.
Суммирование отпадает, так как в этом случае, задача, в конечном итоге, сводится к "привязке" к нескольким значениям констант, практически реализовать которую очень затруднительно.
Инкремент, на первый взгляд, для операции суммирования, приемлем, но в этом случае, результат умножения будет отличаться от истинного.
Чтобы проверить правильность этого высказывания, давайте немного "погоняем нули с единицами на бумаге".
Умножим, например, число **.194** (**11000010**) на **8** (2x2x2), применив, для переносов в старший байт, команды инкремента.

```
194      00000000 11000010 ; множимое, в бите C - ноль.
388      00000001 10000100 ; в бит C "пришла" 1 из старшего бита младшего байта,
                             следовательно, инкремент старшего байта.
521      00000010 00001001 ; из бита C "ушла" 1 в младший бит младшего байта,
                             в бит C "пришла" 1 из старшего бита младшего байта,
                             следовательно, инкремент старшего байта.
531      00000010 00010011 ; из бита C "ушла" 1 в младший бит младшего байта,
                             в бит C "пришел" 0 из старшего бита младшего байта,
                             следовательно, инкремента старшего байта нет.
```

В идеале, должен получиться результат **194x8=.1552**, а получилось **.531**.
Вывод: при организации умножения посредством последовательных, "многоступенчатых" процедур умножения на 2, операции инкремента, для организации переносов в старший байт, применять нельзя.
Проанализировав результат (нужно **.1552**, а получилось **.531**), можно прийти к выводу, что при формировании нескольких переносов, наращивание значений чисел, формируемых в регистре **RegH**, должно происходить значительно бОльшими "темпами", чем в случае применения операций инкремента.
Исходя из этого, на "роль палочки - выручалочки претендует" все та же операция циклического сдвига влево, только применяемая по отношению к содержимому старшего байта.
И в самом деле, она эквивалентна операции умножения на 2, что обеспечивает более высокие "темпы прироста", чем при применении операции инкремента.
Таким образом и "родилась" группа команд умножения на 2, которые можно последовательно ("цепочкой") соединять между собой.
Посмотрите в текст программы.
Вы видите "цепочку" из трех групп команд умножения на 2 (умножение на 8).
Может возникнуть вопрос: "Как два байта делят между собой один бит флага **C**, ведь они оба работают с ним?"
А так и работают - по очереди.

Чтобы это было наглядно, давайте умножим на **8** все то же число **.194** (**11000010**), только с использованием, для переносов в старший байт, не команды **INCF**, а команды **RLF**.

```
194      00000000 11000010 ; множимое, в бите C - ноль.
388      00000001 10000100 ; сдвиг в младшем байте: 0 из бита C "уходит" в
                            младший бит младшего байта, сдвигая все биты влево,
                            1 старшего бита младшего байта "приходит" в бит C.
                            сдвиг в старшем байте: 1 из бита C "уходит" в
                            младший бит старшего байта, сдвигая все биты влево,
                            0 старшего бита старшего байта "приходит" в бит C.
776      00000011 00001000 ; сдвиг в младшем байте: 0 из бита C "уходит" в
                            младший бит младшего байта, сдвигая все биты влево,
                            1 старшего бита младшего байта "приходит" в бит C.
                            сдвиг в старшем байте: 1 из бита C "уходит" в
                            младший бит старшего байта, сдвигая все биты влево,
                            0 старшего бита старшего байта "приходит" в бит C.
1552     00000110 00010000 ; сдвиг в младшем байте: 0 из бита C "уходит" в
                            младший бит младшего байта, сдвигая все биты влево,
                            0 старшего бита младшего байта "приходит" в бит C.
                            сдвиг в старшем байте: 0 из бита C "уходит" в
                            младший бит старшего байта, сдвигая все биты влево,
                            0 старшего бита старшего байта "приходит" в бит C.
```

Вывод: **получен верный результат умножения числа .194 на .8, равный .1552.**
Если анализировать изменения, происходящие с 16-битным числом (2 байта в комплексе), то в его пределах, происходят все те же циклические сдвиги влево, что свидетельствует о том, что посредством "цепочки" групп команд умножения на 2, удалось "привязать" младший байт к старшему таким образом, что конечный результат умножения в точности будет равен 2 в степени N (где N - число групп команд умножения на 2).
Проще говоря, если представить себе некий "барьер" между старшим битом младшего байта и младшим битом старшего байта, препятствующий получению истинного результата умножения на 8, то в случае применения, упомянутой выше, "цепочки" групп команд умножения на 2, этот "барьер разрушается".
В результате, циклические сдвиги влево происходят в "пределах" 16-ти битов (2-х байтов). То есть, два 8-битных регистра "полноценно" объединились в один 16-битный регистр.
Вот Вам и достаточно немудреная суть работы всей "цепочки" (это относится и к "цепочке" групп команд, и к "цепочке" рассуждений).
Для того чтобы закрыть тему умножения на 2 в степени N, остается только выяснить, как происходит взаимодействие групп команд умножения на 2 между собой.
Посмотрите в текст программы.
При опросе бита флага **C**, применяется бит-ориентированная команда ветвления **btfsc**.
Если флаг **C** поднят, то рабочая точка программы переходит на команду **rlf RegH,F**, а после ее исполнения, на первую команду следующей группы команд умножения на 2 (**rlf RegL,F**).
Это первый сценарий.
Если флаг **C** опущен, то рабочая точка программы "перескакивает" команду **rlf RegH,F** (вспоминайте про "виртуальный" **NOP**) и устанавливается на первой команде следующей группы команд умножения на 2 (**rlf RegL,F**).
Это - второй сценарий.
Разница между этими двумя сценариями очень простая: в первом сценарии, команда **rlf RegH,F** исполняется, а во втором, не исполняется (вместо нее исполняется "виртуальный" **NOP**).
В зависимости от величины множителя и выбранного способа его разложения, количество групп команд умножения на 2 может быть различным.
Например, если требуется 2 группы команд умножения на 2, то из текста программы нужно убрать одну группу команд умножения на 2, а если требуется 4, 5 или более, то в текст программы нужно добавить одну, две или более группы команд умножения на 2 соответственно.
После "отработки" команд умножения на 8, рабочая точка программы устанавливается на первую команду первой группы команд суммирования результата умножения на 8 и множимого.
Вот здесь-то содержимое регистра **W** (множимое) и становится востребованным.

Во время исполнения групп команд умножения на 2, регистр **W** не задействовался, и поэтому, на момент начала исполнения групп команд суммирования, в нем находится, записанное в него ранее, значение множимого (в данном случае, **.194**).
Работу этих групп команд я объяснял ранее, так что повторяться не буду.
Происходит следующее:

```
        В аккумуляторе (W) предварительно записано множимое .194 (11000010).
1552      00000110 00010000 ; результат умножения на 8.
1746      00000110 11010010 ; результат 1-го суммирования. В младшем байте -
                              число 16+194=210. Инкремента содержимого старшего
                              байта нет.
1940      00000111 10010100 ; результат 2-го суммирования. В младшем байте -
                              число 210+194=404-256=148. Инкремент содержимого
                              старшего байта (увеличение числа .06 на 1).
```

Итог: **194x8+194+194=194x10=.1940**, что и требовалось получить.
Вместо **.194**, можете использовать любое другое значение множимого (из числового диапазона байта). Вы получите произведение, в 10 раз большее, чем множимое.
Вопрос: "Можно ли реализовать ПП умножения без использования циклических сдвигов влево, а только на основе последовательных суммирований множимого, количество раз, равное множителю"?
Ответ: конечно же можно, но в этом случае, объем подпрограммы увеличится, и при прочих, равных условиях, она будет отрабатываться медленнее.
Каков может быть диапазон используемых множителей?
В рассматриваемом случае (двухбайтный регистр), он такой же, как и диапазон используемых множимых (**.00** ... **.255**), с учетом того, что редко когда возникает практическая необходимость умножать на **0** или **1**.
Если нужно умножить на **.10**, например, число **.10000** (результат умножения больше чем **.65535**), то необходимо "ввести в эксплуатацию" еще один (третий) регистр, причем число **.10000** предварительно будет записываться в два регистра (L и M), а байт регистра H и биты **№ 6** и **7** регистра M предварительно нужно сбросить в **0**.
В этом случае, в связи с необходимостью обслуживания работы дополнительного регистра, в группе команд умножения на 2, количество команд увеличится.
Из сказанного можно сделать следующий практический вывод: если есть готовая подпрограмма, то ее можно "выпотрошить" в симуляторе, но если речь идет о создании подпрограммы, то в ходе ее составления, особо надеяться на помощь симулятора не стОит. Такого рода работа делается, в основном, "в мозгах и на бумаге".
Только после того, как "родится" подпрограмма или какая-то функционально завершенная ее часть (пусть даже и "корявая"), для оценки ее "жизнеспособности", можно (и нужно) задействовать симулятор.
Вывод: "на симулятор надейся, а сам не плошай".
Примерно то же самое можно сказать и о "бесплатном сыре" в виде "готовых к употреблению" программ и подпрограмм.
На практике часто возникает потребность в их "модернизациях" под свои задумки, и если перед этим, как следует не "погонять" нули с единицами в стандартных (классических) подпрограммах (в первую очередь, в голове и на бумаге, а затем и в симуляторе), то можно банально "попасть в мышеловку" своей некомпетентности.
В принципе, программирование это такая же работа, как и любая другая, только связанная с переработкой большого массива информации, изначально "сваленной" в большую и привлекательную "кучу".
Кто сумеет "расчленить" этот массив на функционально законченные части, разобраться с каждой из них и "разложить их по полочкам" так, чтобы они не "наезжали" друг на друга и чтобы было понятно "что где лежит", тот в этой "битве" и будет победителем.

Отследить процесс циклических сдвигов, с задействованием бита **C** регистра **STATUS**, в одно, двух, трех и четырехбайтном регистрах, можно в программе, разработанной **Николаем Маровым**.
Она называется shift_c.exe и находится в папке *"Программы"*.

Дополнительно

Пример поиска и устранения ошибки в программе.

Пришло время объяснить/показать, что такое "бесплатный сыр" и каковы последствия его "употребления".
Речь пойдет об ошибках.
Они допускаются при составлении программ.
Даже самые многоопытные программисты, хотя и не часто, но все-таки их допускают.
Ошибки могут быть и в "чужих" программах.
Часть ошибок можно устранить с помощью симулятора, но самыми сложными ошибками являются ошибки функционального характера: ассемблирование прошло успешно, но программа или не работает вообще, или работает не так, как нужно.
Это встречается сплошь и рядом, и составление редко какой, более или менее сложной программы, обходится без ошибок.
Начинающие программисты, по определению, будут "плодить" их в больших количествах.
Если относиться к ним трагически, то на своем занятии программированием можно смело ставить большой и жирный крест.
Таким образом, для психологической защиты от этой неизбежной "беды", нужно присвоить ей статус "полезности".
И это сделать достаточно просто, так как любая ошибка учит.
Такой довод настолько "железобетонен", что с ним трудно спорить.
Если ошибка обнаружена и исправлена своими силами, то положительный эффект от этого, колоссальный.
Эффективная работа над ошибками, как это не странно звучит, - одна из главных составляющих успеха. Этому тоже нужно учиться.
Исходя из "вышележащих" соображений, при работе над этим разделом, я сознательно "заложил" в текст программы **Rlf.asm** ошибку функционального характера и как рыбак, стал ждать, когда же кто-нибудь "клюнет" (прошу не считать меня "садистом". Примерно по той же причине, по которой нельзя считать "садистом" хирурга).
Эту ошибку не только обнаружил, но и исправил **Марсель Валеев** из г.Глазов.
Думаю, что Марсель не будет на меня в обиде, если я опубликую выдержки из его писем.

Как человек упертый и настойчивый, я тщательно изучаю каждую главу Вашего учебника и, как правило, разрабатываю уже свои программы под свои разработки, опираясь на знания, полученные от вас.
В главе 14 вы привели пример использования команды сдвига в программе **rlf.asm**: умножение на 10 числа от **0** до **255**.
Я встроил ее в **multi.asm**, вставил перед ней пару команд закладки множимого и начал обкатывать, так, любопытства ради.
При умножении выборочных чисел от **2** до **127** проблем не возникло. Результаты сходились.
255 она также умножает без ошибок. Получается **2550**.
При умножении **194** на **8**, получилось **784**.
Далее я попробовал умножить на **8** число **200** и получил **832**.
В результате я выяснил, что сбой происходит при третьем умножении на 2.
Число **800** на **2** умножается неправильно! Сдвиг влево, в старшем байте, не происходит.

Вот Вам и практический пример эффективности использования методики "врезки" подпрограммы в другую "полноценную" программу, о которой говорилось ранее.
Ошибка обнаружена и "локализована".
Далее, как это и положено, идет "раскрутка":

... нужно позаботиться о сдвиге в регистре **RegH**.
В начале я стал вставлять различные подпрограммки ветвления, но это, в конечном итоге, привело к громоздкости.

В конце концов меня "осенило", и я выкинул из начального текста все команды ветвления, которые опрашивают флаг **C** в трех последовательных умножениях на 2, и получилась красивая подпрограмма, которая великолепно умножает все числа в промежутке от **0** до **255**.

Нет вопросов. Оценка "отлично".
А теперь давайте детально разберемся, почему, при умножении числа **.194** или **.200** на **8**, ошибка происходит именно при третьем умножении на **2**, и почему наличие команд ветвления **btfsc Status,C**, в операциях умножения на **2**, является функциональной ошибкой.
В соответствии с описанным выше алгоритмом операции умножения на **10**, в пределах одной операции умножения на **2**, должны последовательно происходить два циклических сдвига влево, в порядке:
- сдвиг содержимого регистра младшего разряда **RegL**,
- сдвиг содержимого регистра старшего разряда **RegH**,

вне зависимости от величины содержимого этих регистров.
Сдвиг содержимого регистра **RegL** происходит всегда, а наличие сдвига содержимого регистра **RegH** поставлено в зависимость от состояния флага переноса **C** (**btfsc Status,C**).
Это означает то, что в зависимости от значения старшего бита регистра младшего разряда **RegL** на момент начала сдвига содержимого регистра **RegH** (после отработки команды **btfsc Status,C**), сдвиг содержимого регистра **RegH** может происходить, а может и не происходить (два сценария).
Детализируем.
Если после исполнения циклического сдвига содержимого регистра **RegL**, в бит **C** "уходит" единица (из старшего бита байта регистра **RegL**), то в дальнейшем, произойдет сдвиг содержимого регистра **RegH** (см. логику работы команды **btfsc**), а если в бит **C** "уйдет" ноль, то сдвига содержимого регистра **RegH** произведено не будет, так как в соответствии с логикой работы команды **btfsc**, команда **rlf RegH,F** будет проигнорирована (вместо нее, будет исполнен "виртуальный" **NOP**).
Это полностью объясняет все то, что наблюдал Марсель при "прогонке", в симуляторе, подпрограммы умножения на 10.
Почему ошибка возникла только на третьем умножении на 2, а два предшествующих умножения были выполнены верно?
Переводим число **.194** и **.200** в бинарный вид: **11000010** и **11001000** соответственно.
Смотрим на биты **№№ 6** и **7**.
И в одном, и в другом, по единице, которые, при первых двух умножениях, будут "переходить" в бит **C**, состояние которого анализируется командой **btfsc Status,C**.
В этих случаях, последующий сдвиг содержимого регистра **RegH** производиться будет.
А вот бит **№ 5** (третий слева) имеет значение **0** и именно он (ноль) "перейдет" в бит **C** на третьей операции умножения на **2**, и заблокирует сдвиг содержимого регистра **RegH** (содержимое регистра **RegL** сдвинется, а сдвига содержимого регистра **RegH** не произойдет), что и имеет место быть.
Для числа **.194**.
Биты **№№ 6** и **7** числа **.194** равны **1**, следовательно, первые две операции умножения на 2 пройдут без ошибок: получается **194x2x2=776**.
Бит **№ 5** числа **.194** равен **0**, следовательно, произойдет сдвиг только содержимого регистра **RegL**, а сдвиг содержимого регистра **RegH** будет заблокирован.
Вот как это выглядит:
- результат 2-й операции x2: **00000011 00001000** - число **.776**.
- результат 3-й операции x2: **00000011 00010000** - число **.784** вместо **194x8=.1552**

(см. выдержку из письма Марселя).
То же самое можно "проделать" и с числом **.200**.
Получится **.832** вместо **200x8=.1600**.
В случаях умножения, на **8**, этих чисел (**.194** или **.200**), еще "повезло", так как ошибка происходит только на третьем умножении на **2**, а если умножить на **8**, например, число **.131** (**10000011**), то ошибочным будет уже второе (а также и третье) умножение на **2**.
Таким образом, наличие в программе описанной выше функциональной ошибки, приводит к тому, что числовой диапазон байта (**.0** … **.255**) как бы "разбивается на зоны", в пределах которых, умножение на **8** может происходить безошибочно или с ошибками.

"Зоны" безошибочной работы:

1. До **.31** включительно: биты **№№ 7,6,5** равны **000**.
В течение всех 3-х операций умножения на 2, для получения верного результата умножения на **8**, достаточно сдвигов в пределах младшего байта (**RegL**), которые гарантированно происходят, и переносов в старший байт (**RegH**) не требуется.
2. От **.32** до **.63** включительно: биты **№№ 7,6,5** равны **001**.
Первые 2 сдвига происходят в пределах младшего байта (**RegL**).
То есть, для получения верного результата, переносов в старший байт (RegH) не требуется.
После этого, на 3-м сдвиге, происходит безошибочный перенос из старшего бита байта регистра **RegL** в младший бит байта регистра **RegH** (через бит **C** регистра **Status**).
3. От **.96** до **.127** включительно: биты **№№ 7,6,5** равны **011**.
Первый сдвиг происходит в пределах **RegL**, а при 2-м и 3-м сдвигах, происходят безошибочные переносы.
4. От **.224** до **.255** включительно: биты **№№ 7,6,5** равны **111**.
При всех трех сдвигах, происходят безошибочные переносы.

"Зоны" работы с ошибками:

1. От **.64** до **.95** включительно: биты **№№ 7,6,5** равны **010**.
При 1-м сдвиге, ошибки не будет, так как переноса в **RegH** не требуется, при 2-м сдвиге, произойдет безошибочный перенос, так как бит **№ 6** равен **1**, а при 3-м сдвиге произойдет ошибка, так как бит **№ 6**, равный нулю, заблокирует сдвиг содержимого регистра **RegH**.
2. От **.128** до **.159** включительно: биты **№№ 7,6,5** равны **100**.
При 1-м сдвиге, произойдет безошибочный перенос, при 2-м и 3-м сдвигах, произойдут ошибки, так как биты **№№ 6** и **5**, равные нулю, заблокируют оба сдвига содержимого регистра **RegH**.
3. От **.160** до **.191** включительно: биты **№№ 7,6,5** равны **101**.
Здесь ошибочной будет 2-я операция x2.
4. От **.192** до **.223** включительно: биты **№№ 7,6,5** равны **110**.
Здесь ошибочной будет 3-я операция x2.

Таким образом, программа **Rlf.asm** является как бы "частично рабочей".
Для того чтобы сделать ее полностью рабочей, нужно изъять из ее текста все три команды **btfsc Status,C**.
В этом случае, будет происходить гарантированный сдвиг содержимого регистра **RegH**, не зависящий от состояния флага **C**.
Это вовсе не означает то, что при создании других алгоритмов умножения, команды ветвления вообще можно "выкинуть на помойку".
Они могут быть необходимы в тех случаях, когда числа из различных "числовых зон", необходимо умножить на различные множители, но это уже более сложная задача, которой лучше заняться только после приобретения достаточных знаний и опыта.
Итак, после детального разбирательства, выяснилось, что при умножении на **8**,
- в "зоне" чисел до **.127** могут происходить ошибки (от **.64** до **.95**),
- в "зоне" чисел от **.128** до .**255** может происходить безошибочная работа (от **.224** до **.255**).

Что дает такое "копание"?
Кроме очень эффективного усвоения информации и приобретения практических навыков, оно дает, подкрепленную реальными делами, уверенность в своих силах, причем эта уверенность относится к самой высокой, "бойцовской" категории.
Если попытаться, в общем виде, сформулировать методику поиска и устранения функциональных ошибок в программах, то это будет выглядеть примерно так:
При отладке программы в симуляторе выясняется, что она работает не так как нужно.
В нем же (в симуляторе) необходимо "локализовать" участок программы, на котором эта ошибка проявляется (максимально сузить "сектор поиска").
Далее, фиксируется несоответствие между тем, что должно быть и реальным результатом (например, как в нашем случае: **194x8=.1552**, а получилось **.784**).
Всё. Симулятор в сторону.
"Принимаем позу мыслителя", готовим литр кофе и пачку сигарет, "включаем мозги" и пытаемся понять, почему получен именно такой результат?
Если по ходу разбирательств, возникает необходимость в дополнительной информации, то опять задействуется симулятор, который, таким образом, является всего-лишь

"поставщиком" необходимой для анализа, исходной информации, но не "панацеей от всех бед".
Все основные "дела" должны "делаться" в мозгах.
Лентяи сразу же "выпадают в осадок", а "выживают" только упертые и въедливые, к которым, вне всякого сомнения, относятся такие люди как Марсель.
В идеале, программист вообще должен обходиться без помощи симулятора, работая только с текстом программы.
Примечание: никаких исправлений в текст программы **Rlf.asm** и в ее описание я вносить не буду, а иначе уничтожается предмет разбирательств.

15. Введение в принцип построения подпрограммы динамической индикации. Косвенная адресация.

А теперь, имея некоторый опыт и навыки, давайте разберемся, как осуществить вывод данных на индикацию, используя принцип динамической индикации.
Совершенно понятно, что устройство, в котором реализовано только это, практической ценности не представляет.
Оно представляет ценность только тогда, когда "вмонтировано" в более сложное устройство.
Функция – визуальное отображение результата/результатов работы всего устройства.
Динамическая индикация организуется тогда, когда линейка состоит из нескольких 7-сегментных индикаторов (знакомест).
Почему, когда речь идет о м/контроллерах, к которым нужно "пристегнуть" несколько 7-сегментных индикаторов, выбирается именно динамический способ индикации, а не "статический", который применялся "на заре" цифровой техники (вспомните про индикаторы, с "кучей" подключенных к ним проводов и "кучей" микросхем, обслуживающих их работу. У меня такая "каракатица" есть. В кладовке лежит. Выбрасывать жалко…)?
Объяснение простое: на эту "старушку", не хватит выводов портов даже достаточно "крутого" м/контроллера.
А если и хватит, то "овчинка выделки не стоит", ведь перейдя на динамическую индикацию, можно получить тот же результат, но гораздо более дешевым способом.
8-выводной порт (порты), которым "укомплектована" значительная часть ПИКов (но не все типы), вполне подходит для управления 7-сегментными индикаторами.
PIC16F84A имеет 2 порта.
В порте B, можно использовать все 8 выводов.
В порте A, можно использовать 5 выводов.
Порт B, "в полном составе", выделяется под управление сегментами 7-сегментных индикаторов (включая и запятую), а функция последовательной активации знакомест передается в порт A.
Рассмотрим случай организации динамической индикации для линейки 7-сегментных индикаторов, состоящей из 8-ми знакомест.
Подпрограмма динамической индикации, о которой будет рассказано, достаточно универсальна, то есть, она может быть "врезана/встроена" в другие программы.
Разрядность можно будет уменьшить (просто) или увеличить (сложнее).
Я об этом расскажу.
Советую Вам как следует разобраться с тем, что Вы прочитаете.
Это очень востребовано.
Лично я, так и не встретил хотя бы более-менее внятного объяснения работы подобного рода подпрограммы, не говоря уж о какой-то универсальной "заготовке", рассчитанной на начинающих (и не только).
"Потрошить" подпрограмму динамической индикации будем в 2 этапа:

- ***на 1-м этапе, сформируем "скелет",***
- ***на 2-м этапе, "нарастим на нем мясо".***

По идее, при "разборках" с подпрограммами, нужно "отталкиваться" от ее блок - схемы.
В данном случае, я совместил текст подпрограммы с ее блок – схемой.
Это позволяет, не "распыляя" внимания, отслеживать и то, и другое, в одном месте (в тексте подпрограммы).
По ходу дела, разберемся с косвенной адресацией, и я расскажу о "хитром" способе записи константы в регистр.

Первый этап

Предмет разбирательства представляет собой текст программы с названием **Fsr.asm**, в которую "врезана" подпрограмма динамической индикации, обслуживающая линейку из 8-ми 7-сегментных индикаторов с **общим катодом**.
Это означает то, что внутри линейки, аноды светодиодов всех одноименных секторов запараллелены.
Таким образом, получается 8 "проволочин", которые, через гасящие резисторы, подключаются к 8-ми выводам порта B.

8 выводов общих катодов индикаторов подключаются к 8-ми выходам дешифратора 555ИД7 (подробнее об этом → ниже).

Файл программы называется Fsr.asm (находится в папке ***"Тексты программ"***).

Это выглядит так:

```
;*******************************************************************************
;     ПРИМЕР ОРГАНИЗАЦИИ ДИНАМИЧЕСКОЙ ИНДИКАЦИИ С ИСПОЛЬЗОВАНИЕМ
;          КОСВЕННОЙ АДРЕСАЦИИ ПРИ ОБРАЩЕНИИ К ТАБЛИЦЕ ДАННЫХ.
;*******************************************************************************
;                                  "ШАПКА ПРОГРАММЫ"
;*******************************************************************************
; Fsr.asm   Универсальная группа подпрограмм 8-разрядной динамической индикации
;.....................................................................
;===============================================================================
; Определение положения регистров специального назначения.
;===============================================================================
Indf        equ         00h         ; Регистр INDF.
PC          equ         02h         ; Регистр счетчика команд.
Status      equ         03h         ; Регистр Status.
FSR         equ         04h         ; Регистр FSR.
;.....................................................................
;.....................................................................
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
LED0        equ         10h         ; Регистр 1-го 7-сегментного индикатора.
LED1        equ         11h         ; ------- 2-го -------------------------
LED2        equ         12h         ; ------- 3-го -------------------------
LED3        equ         13h         ; ------- 4-го -------------------------
LED4        equ         14h         ; ------- 5-го -------------------------
LED5        equ         15h         ; ------- 6-го -------------------------
LED6        equ         16h         ; ------- 7-го -------------------------
LED7        equ         17h         ; ------- 8-го -------------------------
Index       equ         0Ch         ; Регистр счетчика количества
                                    ; малых колец индикации.
Count       equ         0Dh         ; Регистр счетчика количества
                                    ; больших колец индикации.
;.....................................................................
;.....................................................................
;===============================================================================
; Определение места размещения результатов операций.
;===============================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;===============================================================================
; Присваивание битам названий.
;===============================================================================
Z           equ         2           ; Флаг нулевого результата.
;.....................................................................
;.....................................................................
;===============================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*******************************************************************************



;*******************************************************************************
;                                РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;*******************************************************************************
; Подготовительные операции.
;-------------------------------------------------------------------------------
```

```
START       .....................................
            .....................................
;-------------------------------------------------------------------------------
; Группа подпрограмм преобразования двоичных чисел в двоично-десятичные.
;-------------------------------------------------------------------------------
Bin2_10     .....................................
            .....................................
.......     .....................................
            .....................................
.......     .....................................
            .....................................
На данный момент, регистры LED0 ... LED7 заполнены двоично-десятичными числами,
которые необходимо вывести на индикацию (отобразить) в линейку из 8-ми
7-сегментных индикаторов.
На момент начала группы подпрограмм динамической индикации, все прерывания должны
быть запрещены, выводы 8-разрядного порта, к которому подключена линейка должны
быть настроены на работу "на выход", работа должна происходить в нулевом банке.
;-------------------------------------------------------------------------------
; Подготовка счетчика количества малых колец индикации Index к началу полного
; цикла динамической индикации.
;-------------------------------------------------------------------------------
            clrf        Index       ; Сброс в 0 содержимого счетчика
                                    ; малых колец индикации Index.
;-------------------------------------------------------------------------------
; Предварительная закладка количества больших колец индикации, которое нужно
; "пройти" за один полный цикл динамической индикации в регистр Count.
;-------------------------------------------------------------------------------
            movlw       X           ; Запись константы X (количество больших
                                    ; колец индикации, задается программистом)
                                    ; в регистр W.
            movwf       Count       ; Копирование содержимого регистра W в
                                    ; регистр счетчика количества больших колец
                                    ; индикации Count.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Использование косвенной адресации при работе с таблицей данных.
;===============================================================================
; Подготовка к косвенной адресации: запись в регистр W адреса регистра младшего
; разряда линейки 7-сегментных индикаторов ("привязка" к 7-сегментному
; индикатору, с активации которого начинается полный цикл первого большого
; кольца индикации).
;-------------------------------------------------------------------------------
CYCLE       movlw       LED0        ; Запись в регистр W адреса регистра LED0.
            addwf       Index,W     ; Увеличение адреса регистра LED0 на величину
                                    ; числа, записанного в регистре счетчика
                                    ; количества малых колец индикации Index,
                                    ; С сохранением результата в регистре W.
;-------------------------------------------------------------------------------
; Косвенная адресация.
;-------------------------------------------------------------------------------
            movwf       FSR         ; Копирование содержимого регистра W
                                    ; в регистр FSR.
            movf        Indf,W      ; Копирование содержимого регистра с адресом,
                                    ; записанным в регистре FSR, в регистр W.
            call        TABLE       ; Условный переход (адрес следующей команды
                                    ; закладывается в стек) в ПП TABLE.
;---> Возврат по стеку из ПП TABLE.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Группа команд установки запятой.
;-------------------------------------------------------------------------------
            ....................
            ....................
;-------------------------------------------------------------------------------
; Группа команд формирования адресного кода управления дешифратором.
;-------------------------------------------------------------------------------
            ....................
            ....................
```

```
;-------------------------------------------------------------------------
; Группа команд вывода десятичной цифры на индикацию (в порт).
;-------------------------------------------------------------------------
.....       ......................   Начало визуального отображения цифры в
            ......................   одном из 7-сегментных индикаторов (зависит
            ......................   от адресного кода управления дешифратором).
;-------------------------------------------------------------------------
; Группа команд задержки, определяющей время нахождения одного 7-сегментного
; индикатора в активном состоянии (определяющей время прохождения малого кольца
; индикации).
;-------------------------------------------------------------------------
.....       ......................
            ......................
;-------------------------------------------------------------------------
; Увеличение на 1 содержимого счетчика количества малых колец индикации Index с
; последующей проверкой результата инкремента на равенство (или нет) числу .08.
;-------------------------------------------------------------------------
            incf        Index,F     ; Увеличение на 1 содержимого регистра Index
                                    ; с сохранением результата в нем же.
            movlw       .08         ; Запись в регистр W константы .08.
            bcf         Status,Z    ; Сброс флага нулевого результата Z.
            subwf       Index,W     ; Вычесть из содержимого регистра Index
                                    ; содержимое регистра W, с сохранением
                                    ; результата в регистре W.
            btfss       Status,Z    ; Результат операции вычитания равен
                                    ; или нет нулю?
            goto        CYCLE       ; Если не =0 (в регистре Index - число не
                                    ; равное 8), то переход к циклу активизации
                                    ; следующего по старшинству 7-сегментного
                                    ; индикатора (переход на новое малое кольцо
                                    ; индикации, то есть, в ПП CYCLE).
                                    ; Если =0 (в регистре Index - число равное
                                    ; 8), то программа исполняется далее.
;-------------------------------------------------------------------------
; Начало перехода на новое большое кольцо индикации после того, как
; последовательно активизируются все 8 7-сегментных индикатора линейки
; (после прохождения 8-ми малых колец индикации).
;-------------------------------------------------------------------------
            nop                     ; Выравнивающий NOP.
            clrf        Index       ; Сброс в 0 содержимого регистра Index.
;-------------------------------------------------------------------------
; Уменьшение на 1 содержимого счетчика количества больших колец индикации Count.
;-------------------------------------------------------------------------
            decfsz      Count,F     ; Декремент содержимого счетчика количества
                                    ; больших колец индикации Count,
                                    ; с сохранением результата в нем же.
            goto        CYCLE       ; Если результат декремента не=0, то переход
                                    ; в ПП CYCLE
                                    ; (переход на новое большое кольцо индикации)
                                    ; Если результат декремента =0, то программа
                                    ; исполняется далее (переход на новый полный
                                    ; цикл динамической индикации).
            nop                     ; Выравнивающий NOP.
;-------------------------------------------------------------------------
; Группы подпрограмм и команд, осуществляющие различные операции.
;-------------------------------------------------------------------------
            ......................................
            ......................................
            ......................................
            ......................................
            ......................................
            ......................................
```

На данный момент, регистры LED0 ... LED7 заполнены двоичными числами, которые, в дальнейшем, нужно преобразовать в двоично-десятичные и сохранить их все в тех же регистрах LED0 ... LED7.

```
            goto        Bin2_10     ; Безусловный переход в ПП Bin2_10.

;-------------------------------------------------------------------------
;                   Применение вычисляемого перехода.
; (преобразование двоично-десятичного кода в код 7-сегментного индикатора)
;-------------------------------------------------------------------------
TABLE       addwf       PC,F        ; Содержимое счетчика команд PC увеличивается
                                    ; на величину содержимого аккумулятора W.
            retlw       b'00111111' ; ..FEDCBA = 0   Происходит скачек по таблице
            retlw       b'00000110' ; .....CB. = 1   на строку со значением,
            retlw       b'01011011' ; .G.ED.BA = 2   записанным в аккумуляторе,
            retlw       b'01001111' ; .G..DCBA = 3   и далее - возврат по стеку.
            retlw       b'01100110' ; .GF..CB. = 4
            retlw       b'01101101' ; .GF.DC.A = 5
            retlw       b'01111101' ; .GFEDC.A = 6
            retlw       b'00000111' ; .....CBA = 7
            retlw       b'01111111' ; .GFEDCBA = 8
            retlw       b'01101111' ; .GF.DCBA = 9
;*************************************************************************
            end                     ; Конец программы.
```

"Отталкиваемся" от общего принципа динамической индикации, суть которого заключается в последовательной активации 7-сегментных индикаторов (знакомест).
Активация индикатора это, образно выражаясь, его "загорание".
Чаще всего, полный цикл активации начинается с младшего по разрядности, 7-сегментного индикатора (правого в линейке). К этому и "привяжусь".
Некоторое время, он находится в активном состоянии, после чего "гасится" (переводится в пассивное состояние), и тут же активируется следующий индикатор (**более старшего разряда**).
Некоторое время, он находится в активном состоянии, после чего "гасится" (переводится в пассивное состояние), и тут же активируется следующий индикатор (**более старшего разряда**).
Ну и так далее.
До тех пор, пока не "погаснет" индикатор самого старшего разряда.
Сразу же после его "погасания" (перехода в пассивное состояние), текущий, полный цикл активации заканчивается, и начинается следующий, полный цикл активации (активируется самый младший разряд).
Всё повторяется снова. И так происходит много раз.
Вот Вам и типичная "закольцовка", которую и нужно реализовать в группе подпрограмм динамической индикации.
Назовем это кольцо **большим кольцом индикации.**
Время "прохождения", рабочей точкой программы, большого кольца индикации, равно суммарному времени активации всех 8-ми разрядов линейки.
После активации, каждый из индикаторов должен какое-то время "погореть" (какое-то время находиться в активном состоянии), что вполне понятно, а иначе мы просто ничего не увидим.
Таким образом, речь идет о задержке, суть которой - всё та же "закольцовка".
Вот Вам и еще одна "закольцовка", которую нужно реализовать в группе подпрограмм динамической индикации.
Назовем это кольцо **малым кольцом индикации.**
Время "прохождения", рабочей точкой программы, малого кольца индикации, будет в 8 раз меньше времени "прохождения" большого кольца индикации.
Следовательно, время "прохождения", рабочей точкой программы, большого кольца индикации, ставится в зависимость ("привязывается") от времени "прохождения" малого кольца индикации.
Таким образом, интервал времени "прохождения", рабочей точкой программы, малого кольца индикации, является "базовым".
От него зависит и скорость последовательной активации разрядов линейки, и время отработки цикла большого кольца индикации.
Если эту скорость сделать маленькой, то будет визуально наблюдаться процесс так называемых "мерцаний" (визуально ощущаемых активаций).

Естественно, что это не удобно.
Поэтому, скорость последовательной активации знакомест линейки нужно стремиться сделать как можно большей, но в пределах разумного (компромисс).
То есть, эта скорость должна быть такой, чтобы и минимизировать негативные последствия "мерцаний", и существенно не снизить яркость свечения индикаторов.
Далее, возникает вопрос о "рулевом", ведь процессом последовательной активации нужно как-то "рулить".
То есть, должно быть **нечто**, дающее "отмашку" на начало активации следующего разряда линейки (после того, как активация предыдущего разряда линейки закончится).
По здравому разумению, это **нечто** должно быть "встроено" в малое кольцо индикации и иметь "привязку" к количеству задействованных индикаторов линейки.
После окончания активации текущего знакоместа линейки, содержимое этого **нечто** должно измениться на 1 (увеличиться/уменьшиться) таким образом, чтобы на следующем, малом кольце индикации, произошла активация следующего, по старшинству, индикатора линейки.
С учетом сказанного, не трудно догадаться, что "роль этого **нечто** исполняет" регистр общего назначения.
В данном случае, это регистр с названием **Index**, который является счетчиком количества малых колец индикации.
Он "встроен" в цикл малого кольца индикации и инкрементируется в конце интервала времени активации, с последующим анализом его содержимого.
В числовом диапазоне от **.00** до **.07**, рабочая точка программы последовательно "отматывает" **8** малых колец индикации.
В начале **9**-го кольца (число **.08**), текущий, полный цикл большого кольца индикации заканчивается.
Сразу же после этого, начинается формирование "нового", большого кольца индикации, в начале которого, число **.08**, "лежащее" в регистре **Index**, сбрасывается в **0**.
А теперь предположим, что динамическая индикация организована.
Но ведь это же не самоцель. А отображать-то что?
Если рабочая точка программы все время, пока устройство включено, будет "мотать кольца" в подпрограмме динамической индикации, не выходя из нее, то такое устройство и даром не нужно.
Это как в "12-ти стульях": "Все это великолепие разбивалось о вывеску **Штанов нет**".
Вывод: рабочая точка программы, "сделав все дела" в подпрограмме динамической индикации, должна из нее выйти, для того чтобы каким-то образом (вариантов может быть много), в других подпрограммах и группах команд, не относящихся к подпрограмме динамической индикации, сформировать число, которое нужно отобразить в линейке.
В интервале времени между моментом выхода рабочей точки программы из ПП динамической индикации и моментом ее очередного "влёта" в ПП динамической индикации, программа должна сформировать данные, которые нужно вывести на индикацию.
Из этого следует простой вывод: необходимо ограничить "пребывание" рабочей точки программы в ПП динамической индикации.
Вопрос: "Каким образом"?
Ответ напрашивается сам собой. Количество больших циклов индикации нужно "поставить на счетчик".
"Покрутилась" (речь идет о рабочей точке программы) в ПП динамической индикации определенное количество больших циклов индикации?
Будь добра, покинь ее и подготовь то, что нужно вывести на индикацию в следующем, полном цикле программы.
Вот тогда и снова "милости просим".
Советую Вам представить себе рабочую точку программы в виде живого существа, которое, для того чтобы оно не сотворило чего-то нехорошего, нужно постоянно учить уму-разуму и опекать. Срабатывает на все сто (на себе проверил).
Итак, для того чтобы ограничить время пребывания рабочей точки программы в ПП динамической индикации, в качестве счетчика количества больших колец индикации, нужно назначить регистр общего назначения, предварительно (до "влёта" в цикл большого кольца индикации) записать в него константу (заданное количество больших колец индикации) и в конце цикла большого кольца индикации, декрементировать его содержимое.
Если результатом декремента будет являться число отличное от **0**, то рабочая точка программы "улетит" на начало следующего цикла большого кольца индикации.

После того как в счетчике будет обнаружен **0** (заданное количество циклов отработано), отрабатывается сценарий "программа исполняется далее" (выход из ПП динамической индикации).
Подобное уже описывалось ранее, так что ничего нового в этом нет.
Итак, терминология такая:

- малое кольцо динамической индикации,
- большое кольцо динамической индикации,
- полный цикл динамической индикации,
- полный цикл программы.

Почему "полный цикл динамической индикации", а не "полное кольцо динамической индикации"?
Потому, что полный цикл динамической индикации не является кольцом.
В противном случае, рабочая точка программы не сможет выйти из ПП динамической индикации.
По факту, "закольцовка"есть, но только через полный цикл программы.
А теперь, начиная с малого кольца индикации, в общем виде, представьте себе, как именно происходит движение рабочей точки программы, и какие "фигуры высшего пилотажа" она "проделывает".
Если будут затруднения, то еще раз повнимательнее прочитайте то, о чем говорилось выше.
Прежде чем перейти к деталям, осталось выяснить, каким должно быть соотношение времени отработки ***полного цикла динамической индикации***, к времени отработки ***полного цикла программы***.
В идеале, это соотношение должно быть чуть меньше 1 (больше чем 1, оно быть не может).
То есть, величина интервала времени между моментом выхода рабочей точки программы из ПП динамической индикации и моментом начала следующего, полного цикла динамической индикации, должна быть как можно меньшей.
Проще говоря, рабочая точка программы, в интервале времени отработки большей части полного цикла программы, должна находиться в ПП динамической индикации, а всё остальное должно отрабатываться очень быстро.
Дело в том, что, после отработки ПП динамической индикации, и до начала "нового", полного цикла динамической индикации, процесс динамической индикации прерывается.
Не трудно догадаться, что при этом, в линейке будет "высвечиваться" содержимое одного знакоместа, а остальные будут "погашены" (пассивны).
При выходе из ПП динамической индикации, можно "погасить" и это знакоместо, но смысл в этом небольшой, так как это практически не уменьшает эффект "мерцаний".
В любом случае, периодическая "пассивность" подпрограммы динамической индикации приводит к "мерцаниям/морганиям" показаний, с периодом равным интервалу времени отработки полного цикла программы.
Для того чтобы максимально ослабить негативные последствия "мерцаний", необходимо максимально уменьшить время этого "моргания", то есть, стремиться к тому, чтобы та часть программы, которая не относится к ПП динамической индикации, отрабатывалась как можно быстрее.
То есть, в этой "части", в идеале, должны отсутствовать подпрограммы задержек, и количество команд должно быть минимальным.
Первое актуальнее, чем второе.
По причине того, что интервал времени, формируемый даже одной ПП задержки, может многократно превышать интервал времени отработки всех остальных команд, указанного выше, участка программы.
А если ПП задержек несколько и они "массивны" (по суммарному времени отработки)?
Получим "сильнейшие моргания/мерцания".
Вывод в идеале: при разработке программы, в состав которой входит ПП динамической индикации, необходимо сделать так, чтобы в состав "участка пассивности" подпрограммы динамической индикации не входила ни одна ПП задержки, и он был "линейным и коротким".
На практике, такое возможно сделать только в простых программах.
По мере возрастания их сложности, программист вынужден будет идти на компромисс.
И еще одна особенность, связанная с ПП динамической индикации.
Предположим, что разрабатывается измерительное устройство, с отображением результатов измерений в линейке, состоящей из 7-сегментных индикаторов. Например, частотомер.

Следовательно, нужно сформировать, довольно-таки значительный, интервал времени измерения.
Если ПП динамической индикации будет работать отдельно от группы программ формирования интервала времени измерения, то соотношение времени отработки полного цикла динамической индикации, к времени отработки полного цикла программы, очень сильно ухудшится, что приведет к сильнейшим "мерцаниям".
Вопрос: "Как выйти из этого затруднительного положения?"
Ответ: цикл динамической индикации должен отрабатываться внутри цикла формирования интервала времени измерения, то есть, входить в его состав.
В этом случае, "и волки сыты и овцы целы" (совмещение функций).
Как это делается?
В сущности, это делается очень просто.
В начале полного цикла динамической индикации, время отработки которого калибровано (не "гуляет"), счетный вход микроконтроллера разблокируется (начало счета), а после окончания отработки полного цикла динамической индикации, блокируется (конец счета).
С таким расчетом, чтобы интервал времени, между моментом разблокировки счетного входа и моментом его блокировки, был в точности равен значению измерительного интервала времени.
Интервал времени отработки полного цикла динамической индикации нужно сделать несколько меньшим, чем минимальный (если их несколько) интервал времени измерения.
Это обусловлено тем, что сразу же после окончания отработки полного цикла динамической индикации и до момента блокировки счетного входа микроконтроллера (окончания счета), нужно произвести точную "доводку" (калибровку) интервала времени измерения до требуемой величины.
В состав ПП динамической индикации должны входить команды ветвления, обеспечивающие, в "границах" полного цикла динамической индикации (но не вне их), соответствующие "закольцовки".
А теперь, после того как сформированы общие представления о принципах работы подпрограмм динамической индикации, займемся деталями.

Работаем с текстом программы **Fsr.asm**

На "шапке" программы останавливаться не буду (ничего нового нет).
В ней "прописано" все то, что используется в рабочей части программы.
ПП **START** - стандартная (подготовительные операции. С учетом специфики разрабатываемого устройства).
Далее следует ПП преобразования двоичных чисел в двоично - десятичные.
Обоснование нужности этой ПП было дано ранее, а "разборки" с ней будут позднее.
Для определенности, нужно "привязаться к начальной точке отсчета".
То есть, дать ответ на вопрос: "Какое знакоместо линейки, после влёта рабочей точки программы, в ПП динамической индикации, нужно активизировать первым?"
Обычно, начинают с самого младшего знакоместа (разряда), что и имеет место быть.
Содержимое **счетчика малых колец индикации** сбрасывается в **0** (**clrf Index**).
Перед "влётом" в малое кольцо индикации, в **счетчик количества больших колец индикации Count**, нужно записать константу.
То есть, количество больших колец индикации, которые должна "отмотать" рабочая точка программы для того, чтобы выйти из полного цикла динамической индикации (по сценарию "программа исполняется далее").
Константа указана в общем виде (**X**).
Ее числовое значение определяет программист.
Далее, по тексту программы, Вы видите общую "точку входа" в малое и большое кольца индикации, которой является первая команда ПП **CYCLE** (**movlw LED0**).
Посмотрите в "шапку" программы.
Вы увидите, что в ней "прописаны" 8 регистров общего назначения **LED0 ... LED7**.
В них, с соблюдением порядка старшинства, хранятся двоично-десятичные числа, которые, в дальнейшем, нужно "протащить" через ПП кодировки и после этого, вывести на индикацию.
Эти числа записываются в них еще до "влёта" рабочей точки программы в ПП динамической индикации (а именно, после отработки ПП преобразования двоичных чисел в двоично - десятичные).

В интервале времени отработки полного цикла динамической индикации, их содержимое не меняется (в них ничего не записывается).
Их содержимое только последовательно считывается.
В тексте программы, Вы видите только обращение к регистру **LED0**.
Куда "испарились" остальные 7 регистров (**LED1 ... LED7**), ведь в тексте программы, прямых обращений к ним нет?
Эти "чудеса" называются **косвенной адресацией**.
Прежде всего, к ней нужно подготовится.
Необходимость этой подготовки обусловлена тем, что после каждого перехода, с предыдущего малого кольца индикации, на последующее, нужно активировать следующее (по старшинству) знакоместо.
То есть, принципом динамической индикации.
Команда **movlw LED0** является "хитрой" командой.
По определению, с ее помощью, в регистр **W**, должна быть записана некая константа, но только где она?
На ее "месте" находится **LED0**.
Вопрос: "Какая же это константа"?
Ответ: а самая что ни на есть натуральная.
Посмотрите в "шапку" программы.
Регистру **LED0**, директивой **equ**, присвоен адрес **10h**.
Это и есть константа.
В этом случае, **MPLAB** "видит" не название регистра, а его адрес.
Таким образом, в регистр **W**, записывается адрес регистра **LED0**.
То есть, командой **movlw LED0** задается "начальная точка отсчета", к которой, через процедуру косвенной адресации, осуществляется "привязка" адресов всех остальных **LED**ов.
Содержимое регистра **Index**, от цикла к циклу малого кольца индикации, будет инкрементироваться, последовательно возрастая от **.00** до **.08**.
Следовательно, если в каждом цикле малого кольца индикации, производить суммирование фиксированной константы **10h** (адрес регистра **LED0**) и содержимого регистра **Index** (**addwf Index,W**), то от цикла к циклу малого кольца индикации, результат суммирования (сохраняется в регистре **W**) будет последовательно увеличиваться от **10h** до **17h**.
Почему не до **18h**?
Потому, что "появление", в регистре **Index**, восьмерки, есть критерий сброса его содержимого в **0**, который происходит в самом начале ее (восьмерки) "появления".
Таким образом, в ходе отработки ПП динамической индикации, в регистре **W**, будет происходить последовательный перебор (по кольцу) адресов регистров **LED0 ... LED7**.
Зачем это нужно и "к какой стенке прислонить" содержимое этих регистров?
Чтобы ответить на этот вопрос, нужно поподробнее разобраться с тем, что называется **косвенной адресацией**.
В части касающейся байт-ориентированных команд (команды работы с константами не в счёт), до сих пор, использовалась только прямая (непосредственная) адресация, при которой команда обращается к названию регистра.
Косвенная адресация есть способ обращения к содержимому регистра, посредством выбора его адреса (название регистра не указывается).
К командам работы с константами это не относится.
Эта нужная, но "темная лошадка" многим попортила нервы, так что имеет большой практический смысл с ней разобраться.
Кроме того, косвенная адресация – "вещь" исключительно полезная, так как очень часто, ее применение позволяет заменить "громоздкие" процедуры на более "компактные" (выигрыш по количеству команд), причем, без изменения функциональности.
Косвенная адресация начинается с копирования (через регистр **W**), из области оперативной памяти, в регистр специального назначения с названием **FSR**, адреса регистра, содержимое которого, в дальнейшем, будет использоваться.
Регистр **FSR** задействуется только при косвенной адресации.
Естественно, что до исполнения команды **movwf FSR**, адрес этого регистра должен быть записан в регистр **W** (см. **movlw LED0**).
В данном случае, в регистр **W**, на самом первом "витке" малого кольца индикации, записывается число **10h**, и далее, в ходе отработки ПП динамической индикации, в регистр **W**, будут последовательно и по кольцу, записываться числа в диапазоне от **10h** до **17h**

(адреса регистров **LED0 ... LED7**).
Сразу же после записи, в регистр **W**, любого из этих чисел, оно копируется, из регистра **W**, в регистр **FSR** (**movwf FSR**).
Это приводит к тому, что по ходу отработки ПП динамической индикации, в регистре **FSR**, происходит последовательный "перебор" адресов (по кольцу) тех регистров, содержимое которых, в дальнейшем, будет использоваться.
Для того чтобы скопировать (считать) содержимое регистра с адресом, указанным в регистре **FSR**, в регистр **W** (для дальнейшего использования), нужно выполнить команду **movf Indf,W**.
То есть, ***обращение к регистру Indf соответствует обращению к содержимому того регистра, адрес которого записан в регистре FSR.***
Для удобства, можно коварно перефразировать так: команда **movf Indf,W** обращается не к "регистру" **Indf** ("чистая подстава", так как он физически не реализован. Просто "кнопка, которая включает действие"), а к содержимому того регистра, адрес которого "лежит" (на момент обращения к **Indf**) в регистре **FSR**.
Регистр специального назначения **Indf**, также, как и регистр специального назначения **FSR**, задействуется только при косвенной адресации.
Оба они находятся в 0-м банке и отображаются в 1-м банке, так что работать с ними можно, "не забивая себе голову" выборами банков.
Итак, в результате работы первых 4-х команд ПП **CYCLE**, на каждом "витке" малого кольца индикации, в регистр **W**, будет по очереди, последовательно копироваться содержимое регистров **LED0 ... LED7**.
Содержимое 8-ми регистров **LED0 ... LED7** представляет собой 8 двоично-десятичных чисел, в диапазоне от **.0** до **.9** (с "привязкой" к порядку старшинства), значения которых, после перекодировки, нужно последовательно отобразить в соответствующих знакоместах.
Это мы уже проходили.
Вспомните ПП **TABLE**, в которой осуществляется перекодировка двоично-десятичного кода в код 7-сегментных индикаторов ("программа" **addwfpc.asm**).
В регистре **W**, находится то, что нужно перекодировать.
Останавливаться на этом я не буду (см. программу **addwfpc.asm**).
Только что мы рассмотрели пример организации ***косвенной адресации с обращением к таблице данных***.
То же самое можно сделать и без использования косвенной адресации, но в этом случае, количество команд возрастет, и весьма существенно.
Для "входа в таблицу", используется команда условного перехода **call**.
После возврата по стеку, программа будет исполняться далее.
В данном случае, ПП **TABLE** размещена в конце программы, но ее можно разместить и в начале программы.
Посмотрите в текст программы **Fsr.asm**.
После команды **call TABLE**, Вы видите "линейный" участок из 4 последовательно исполняемых групп команд.
Детально, их состав и работа будут рассмотрены на 2-м этапе "разборок", а пока, буквально пару слов о них.
Зачем нужна группа команд установки запятой, понятно уже из ее названия.
Группа команд формирования адресного кода управления дешифратором формирует этот самый адресный код.
Ранее, я "привязал" дешифратор к м/схеме 555ИД7, имеющей 3 адресных входа и 8 выходов (да будет так).
В зависимости от адресного кода, на каком-либо одном из 8-ми выходов дешифратора устанавливается **0** (на остальных **1**).
В результате этого, активируется тот 7-сегментный индикатор линейки, к общим катодам которого подключен этот выход.
Возникает вопрос: "Можно ли обойтись без внешнего дешифратора, передав его функции микроконтроллеру?
Ответ: да, можно, но только в случае наличия еще одного (кроме двух задействованных) 8-выводного порта, к которому, в этом случае, могут быть подключены 8 выводов общих катодов 7-сегментных индикаторов линейки.
В **PIC16F84A,** третьего порта нет и поэтому применение внешнего дешифратора необходимо.
Для микроконтроллеров, с количеством портов не менее 3-х, дешифратор можно организовать программно.

В этом случае, в предлагаемую Вашему вниманию "заготовку" программы, на место группы команд формирования адресного кода управления дешифратором, нужно "поставить" группу команд, которая реализует дешифратор.
Группа команд вывода на индикацию символов десятичных цифр также в особых пояснениях не нуждается: тот или иной символ десятичной цифры, в виде перекодированного байта, банально записывается в защелки порта B.
Вывод символа на индикацию будет осуществлен в тот 7-сегментный индикатор, на общих катодах которого дешифратор установил **0**.
Функция группы команд задержки, определяющей время нахождения одного 7-сегментного индикатора в активном состоянии, ясна из названия этой группы команд.
В ней осуществляется подбор числового значения времязадающей константы (констант).
Предположим, что один "виток" малого кольца индикации "отмотан", и произошел переход на следующий его "виток".
Для того чтобы, на этом "витке", обеспечить:
- обработку байта, хранящегося в следующем (по порядку старшинства) **LED**е,
- изменение адресного кода дешифратора, приводящее к активации следующего (по порядку старшинства) знакоместа,

необходимо увеличить, на **1**, содержимое "рулевого" регистра **Index**, которое и происходит после исполнения команды **incf Index,F**.
После инкремента содержимого регистра **Index**, нужно обязательно проверить, а не закончилась ли отработка цикла большого кольца индикации (после этого, нужно переходить на следующий цикл)?
Или по-другому: не установилась ли, в регистре **Index**, восьмерка?
Для этого, организуется проверочная группа команд, с задействованием флага нулевого результата **Z**.
Это мы уже проходили.
Отличие заключается только в том, что вместо команды операции с константой (**sublw**), используется команда операции с содержимым регистра (**subwf**), к которому эта команда обращается.
В регистр **W**, записывается константа **.08**.
Флаг **Z** сбрасывается в **0** (а можно и не сбрасывать, так как это флаг 1-й группы).
Из содержимого регистра **Index**, вычитается содержимое регистра **W** (то есть, число **.08**), и после этого, проверяется состояние флага нулевого результата **Z**.
Результат этой операции будет нулевым только в том случае, если в регистре **Index** будет "лежать" число **.08**.
Если число, "лежащее" в регистре **Index**, отлично от восьмерки, то после проверки состояния флага **Z**, произойдет безусловный переход в ПП **CYCLE**, то есть, на новый "виток" малого кольца индикации, после чего, описанная выше "цепочка событий" повторится, но только будет обрабатываться содержимое следующего, по старшинству, **LED**а.
Если число, "лежащее" в регистре **Index**, будет равно 8-ми, то после проверки состояния флага **Z**, команда **goto CYCLE** будет проигнорирована (вместо нее, исполняется "виртуальный" **nop**), и будет отработан сценарий типа "программа исполняется далее".
Если это так, то выполняется **выравнивающий nop**.
В том смысле, что он устраняет "разнобой" в 1 м.ц. между указанными выше, сценариями ветвления.
И в самом деле, переход рабочей точки программы на 1-ю команду одного сценария, происходит через 2 м.ц. (команда **goto** исполняется за 2 м.ц.), а ее переход на 1-ю команду другого сценария, через 1 м.ц. (команда "виртуального" **NOP**а исполняется за 1 м.ц.).
Если речь идет о "плюс-минус двух лаптях", то выравнивающий **nop** не нужен, но если ПП динамической индикации участвует в формировании калиброванных интервалов времени, то выравнивание необходимо.
Это влияет на стабильность результатов измерений.
В данном случае, "разнобой" не велик, и на него вполне можно "закрыть глаза", но дело не в нем, а в принципе.
А это уже серьёзно, так как речь идет о качестве измерительных устройств.
В данном случае, выравнивающий **nop** только один, но в других случаях, их может потребоваться несколько.

На этот "невзрачный", выравнивающий **nop** я обращаю Ваше внимание по той причине, что в случаях наличия ветвлений, происходящих внутри калиброванных циклов, необходимость в выравнивании возникает достаточно часто.
Только в этом случае, можно сформировать и калиброванный, и высокостабильный интервал времени.
Итак, в случае исполнения сценария "программа исполняется далее", после выравнивающего **NOP**а, исполняется команда **clrf Index**.
То есть, байт "рулевого" регистра **Index** сбрасывается в **0**, что есть подготовка к переходу на следующий цикл большого кольца индикации.
Перед тем, как его осуществить, с целью обеспечения выхода из ПП динамической индикации, необходимо уменьшить, на **1**, содержимое счетчика больших колец индикации **Count** (**decfsz Count,F**).
Команда ветвления **decfsz**, кроме декремента, еще и осуществляет проверку типа "результат декремента равен или не равен нулю?"
Если результат декремента содержимого регистра **Count** не равен нулю, то исполнится команда **goto CYCLE**, и рабочая точка программы "улетит" на начало следующего цикла.
После отработки следующего цикла большого кольца индикации, произойдет следующий декремент содержимого регистра **Count** и т.д.
До тех пор, пока в регистре **Count** не "появится" **0**.
После этого, команда **goto CYCLE** будет проигнорирована (вместо нее, "виртуальный" **NOP**), и далее, будет исполнена команда выравнивающего **NOP**а.
После этого, рабочая точка программы выходит из ПП динамической индикации (завершение полного цикла динамической индикации), и после исполнения групп ПП и команд, осуществляющих различные "полезные" операции, "улетает" (**goto Bin2_10**) на первую команду ПП **Bin2_10**, которую можно считать точкой входа в полный цикл программы.
После этого, всё то, о чем шла речь, будет многократно повторяться до тех пор, пока не будет выключено питание устройства.
Выход на следующий цикл большого кольца индикации происходит сразу же после смены содержимого регистра **Index** с **.07** на **.08**.
"Сразу же" означает то, что этот выход происходит очень быстро, но не "мгновенно".
Посмотрите в текст "программы".
После команды инкремента содержимого регистра **Index**, и до команды сброса его содержимого в **0** (в том случае, если результат инкремента равен **.08**), отрабатывается 6 м.ц.
Таким образом, смена восьмерки на **0** произойдет через 6 м.ц. (в начале 7-го), то есть, быстро.

Второй этап

Теперь разберемся с группами команд, которые в тексте ПП динамической индикации, отмечены прерывистыми линиями из точек, то есть, полностью оформим текст ПП динамической индикации.
Вашему вниманию предоставляется ASM-файл универсальной "заготовки" ПП 8-разрядной динамической индикации, с использованием внешнего дешифратора.

Файл программы называется Dinam.asm (находится в папке ***"Тексты программ"***).

Она выглядит так:

```
;*******************************************************************************
;      ГРУППА ПОДПРОГРАММ 8-РАЗРЯДНОЙ ДИНАМИЧЕСКОЙ ИНДИКАЦИИ С
;                ИСПОЛЬЗОВАНИЕМ ВНЕШНЕГО ДЕШИФРАТОРА.
;*******************************************************************************
;                                 "ШАПКА ПРОГРАММЫ"
;*******************************************************************************
; Dinam.asm Универсальная группа подпрограмм 8-разрядной динамической индикации
;.....................................................
;===============================================================================
; Определение положения регистров специального назначения.
```

```
;=====================================================================
Indf        equ         00h         ; Регистр INDF.
PC          equ         02h         ; Регистр счетчика команд.
Status      equ         03h         ; Регистр Status.
FSR         equ         04h         ; Регистр FSR.
PortA       equ         05h         ; Регистр PortA.
PortB       equ         06h         ; Регистр PortB.
;.....................................................
;.....................................................
;=====================================================================
; Определение названия и положения регистров общего назначения.
;=====================================================================
LED0        equ         10h         ; Регистр 1-го 7-сегментного индикатора.
LED1        equ         11h         ; ------- 2-го ------------------------
LED2        equ         12h         ; ------- 3-го ------------------------
LED3        equ         13h         ; ------- 4-го ------------------------
LED4        equ         14h         ; ------- 5-го ------------------------
LED5        equ         15h         ; ------- 6-го ------------------------
LED6        equ         16h         ; ------- 7-го ------------------------
LED7        equ         17h         ; ------- 8-го ------------------------
Index       equ         0Ch         ; Регистр счетчика количества
                                    ; малых колец индикации.
Count       equ         0Dh         ; Регистр счетчика количества
                                    ; больших колец индикации.
Temp        equ         0Fh         ; Регистр временного хранения данных.
;.....................................................
;.....................................................
;=====================================================================
; Определение места размещения результатов операций.
;=====================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;=====================================================================
; Присваивание битам названий.
;=====================================================================
Z           equ         2           ; Флаг нулевого результата.
;.....................................................
;.....................................................
;=====================================================================
; Присваивание константам названий.
;=====================================================================
Const1      equ         Y1          ; Y1 - значение времязадающей константы
                                    ; "грубо" (до .255). Задается программистом.
Const2      equ         Y2          ; Y2 - значение времязадающей константы
                                    ; "точно" (до .255). Задается программистом.
;=====================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*********************************************************************

;*********************************************************************
;                            РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;*********************************************************************
; Подготовительные операции.
;---------------------------------------------------------------------
;START       ....................................
;           ....................................
;---------------------------------------------------------------------
; Группа подпрограмм преобразования двоичных чисел в двоично-десятичные.
;---------------------------------------------------------------------
;Bin2_10     ....................................
;           ....................................
;.......    ....................................
;           ....................................
;.......    ....................................
;           ....................................
```

```
; На данный момент, регистры LED0 ... LED7 заполнены двоично-десятичными числами,
; которые необходимо вывести на индикацию (отобразить) в линейку из 8-ми
; 7-сегментных индикаторов.
; На момент начала группы подпрограмм динамической индикации, все прерывания
; должны быть запрещены, все выводы порта B и первые 3 вывода порта A должны быть
; настроены на работу "на выход", работа должна происходить в нулевом банке.
;*******************************************************************************
; Подготовка счетчика количества малых колец индикации Index к началу полного
; цикла динамической индикации.
;-------------------------------------------------------------------------------
            clrf        Index       ; Сброс в 0 содержимого счетчика
                                    ; малых колец индикации Index.
;-------------------------------------------------------------------------------
; Предварительная закладка количества больших колец индикации, которое нужно
; "пройти" за один полный цикл динамической индикации в регистр Count.
;-------------------------------------------------------------------------------
            movlw       X           ; Запись константы X (количество больших
                                    ; колец индикации, задается программистом),
                                    ; в регистр W.
            movwf       Count       ; Копирование содержимого регистра W в
                                    ; регистр счетчика количества больших колец
                                    ; индикации Count.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Использование косвенной адресации при работе с таблицей данных.
;===============================================================================
; Подготовка к косвенной адресации: запись в регистр W адреса регистра младшего
; разряда линейки 7-сегментных индикаторов ("привязка" к 7-сегментному
; индикатору, с активации которого начинается полный цикл первого большого
; кольца индикации).
;-------------------------------------------------------------------------------
CYCLE       movlw       LED0        ; Запись в регистр W адреса регистра LED0.
            addwf       Index,W     ; Увеличение адреса регистра LED0 на величину
                                    ; числа, записанного в регистре счетчика
                                    ; количества малых колец индикации Index,
                                    ; С сохранением результата в регистре W.
;-------------------------------------------------------------------------------
; Косвенная адресация.
;-------------------------------------------------------------------------------
            movwf       FSR         ; Копирование содержимого регистра W
                                    ; в регистр FSR.
            movf        Indf,W      ; Копирование содержимого регистра с адресом,
                                    ; записанным в регистре FSR, в регистр W.
            call        TABLE       ; Условный переход (адрес следующей команды
                                    ; закладывается в стек) в ПП TABLE.
;---> Возврат по стеку из ПП TABLE
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Группа команд установки запятой.
;-------------------------------------------------------------------------------
            movwf       Temp        ; Копирование содержимого регистра W (7-
                                    ; сегментные коды индицируемых двоично-
                                    ; десятичных чисел) в регистр Temp.
            movlw       5           ; Запись в регистр W константы .05.
            bsf         Status,Z    ; Поднятие флага нулевого результата Z.
            subwf       Index,W     ; Вычесть содержимое регистра W (число .05)
                                    ; из содержимого регистра Index
                                    ; (числа от .00 до .07).
            btfss       Status,Z    ; Проверка состояния флага Z.
            goto        No_Dot      ; Если флаг Z опущен (результат операции
                                    ; не=0), то переход в ПП No_Dot
                                    ; (запятая не выставляется).
            bsf         Temp,7      ; Если флаг Z поднят (результат операции=0),
                                    ; то установка в единицу 7-го бита
                                    ; (установка запятой) регистра Temp.
;-------------------------------------------------------------------------------
; Группа команд вывода десятичной цифры на индикацию.
;-------------------------------------------------------------------------------
```

```
No_Dot      movf        Temp,W      ; Копирование содержимого регистра Temp (7-
                                    ; сегментные коды индицируемых двоично-
                                    ; десятичных чисел) в регистр W.
            movwf       PortB       ; Копирование содержимого регистра W
                                    ; в 8 защелок порта B.
;-------------------------------------------------------------------------------
; Группа команд формирования адресного кода управления дешифратором.
;-------------------------------------------------------------------------------
            movf        Index,W     ; Копирование содержимого регистра Index
                                    ; в регистр W.
            movwf       PortA       ; Копирование содержимого регистра W в первые
                                    ; 3 защелки порта A (работа "на выход"),
                                    ; управляющие адресными входами внешнего
                                    ; дешифратора.
;-------------------------------------------------------------------------------
; Группа команд задержки, определяющей время нахождения одного 7-сегментного
; индикатора в активном состоянии (определяющей время прохождения малого
; кольца индикации).
; "Грубое" формирование времени полного цикла динамической индикации.
;-------------------------------------------------------------------------------
            movlw       Const1      ; Запись в регистр W константы Y1 (см"шапку")
            movwf       Temp        ; Копирование содержимого регистра W
                                    ; в регистр Temp.
PAUSE       decfsz      Temp,F      ; Декремент содержимого регистра Temp с
                                    ; сохранением результата в нем же.
            goto        PAUSE       ; Если результат декремента не=0,
                                    ; то переход в ПП PAUSE.
                                    ; Если результат декремента =0,
                                    ; то программа исполняется далее.
;-------------------------------------------------------------------------------
; Увеличение на 1 содержимого счетчика количества малых колец индикации Index с
; последующей проверкой результата инкремента на равенство (или нет) числу .08.
;-------------------------------------------------------------------------------
            incf        Index,F     ; Увеличение на 1 содержимого регистра Index
                                    ; с сохранением результата в нем же.
            movlw       .08         ; Запись в регистр W константы .08.
            bcf         Status,Z    ; Сброс флага нулевого результата Z.
            subwf       Index,W     ; Вычесть из содержимого регистра Index
                                    ; содержимое регистра W, с сохранением
                                    ; результата в регистре W.
            btfss       Status,Z    ; Результат операции вычитания равен
                                    ; или нет нулю?
            goto        CYCLE       ; Если не =0 (в регистре Index - число не
                                    ; равное 8), то переход к циклу активизации
                                    ; следующего по старшинству 7-сегментного
                                    ; индикатора (переход на новое малое кольцо
                                    ; индикации, то есть, в ПП CYCLE).
                                    ; Если =0 (в регистре Index - число равное
                                    ; 8), то программа исполняется далее.
;-------------------------------------------------------------------------------
; Начало перехода на новое большое кольцо индикации после того, как
; последовательно активируются все 8 7-сегментных индикатора линейки
; (после прохождения 8-ми малых колец индикации).
;-------------------------------------------------------------------------------
            nop                     ; Выравнивающий NOP.
            clrf        Index       ; Сброс в 0 содержимого регистра Index.
;-------------------------------------------------------------------------------
; Уменьшение на 1 содержимого счетчика количества больших колец индикации Count.
;-------------------------------------------------------------------------------
            decfsz      Count,F     ; Декремент содержимого счетчика количества
                                    ; больших колец индикации Count,
                                    ; с сохранением результата в нем же.
            goto        CYCLE       ; Если результат декремента не=0, то переход
                                    ; в ПП CYCLE
                                    ; (переход на новое большое кольцо индикации)
                                    ; Если результат декремента =0, то программа
```

```
                                    ; исполняется далее (переход на новый полный
                                    ; цикл динамической индикации).
            nop                     ; Выравнивающий NOP.
;=============================================================================
; Группы подпрограмм и команд, осуществляющие различные операции.
;=============================================================================
; "Точное" формирование времени полного цикла динамической индикации (если
; требуется точно калиброванное время полного цикла динамической индикации для
; использования его в качестве измерительного интервала).
;-----------------------------------------------------------------------------
            movlw       Const2      ; Запись в регистр W константы Y2 (см"шапку")
            movwf       Temp        ; Копирование содержимого регистра W
                                    ; в регистр Temp.
PAUSE_1     decfsz      Temp,F      ; Декремент содержимого регистра Temp с
                                    ; сохранением результата в нем же.
            goto        PAUSE_1     ; Если результат декремента не=0,
                                    ; то переход в ПП PAUSE_1.
                                    ; Если результат декремента =0,
                                    ; то программа исполняется далее.
;-----------------------------------------------------------------------------
; Гашение активного разряда линейки.
;-----------------------------------------------------------------------------
            movlw       0           ; Запись в регистр W константы .00.
            movwf       PortB       ; Сброс в 0 всех защелок порта B.
;-----------------------------------------------------------------------------
;           .....................................
;           .....................................
;           .....................................
;           .....................................
;           .....................................
;           .....................................
; На данный момент, регистры LED0 ... LED7 заполнены двоичными числами, которые,
; в дальнейшем, нужно преобразовать в двоично-десятичные и сохранить их все в тех
; же регистрах LED0 ... LED7.
            goto        Bin2_10     ; Безусловный переход в ПП Bin2_10.
;-----------------------------------------------------------------------------
;                   Применение вычисляемого перехода.
; (преобразование двоично-десятичного кода в код 7-сегментного индикатора)
;-----------------------------------------------------------------------------
TABLE       addwf       PC,F        ; Содержимое счетчика команд PC увеличивается
                                    ; на величину содержимого аккумулятора W.
            retlw       b'00111111' ; ..FEDCBA = 0   Происходит скачек по таблице
            retlw       b'00000110' ; .....CB. = 1   на строку со значением,
            retlw       b'01011011' ; .G.ED.BA = 2   записанным в аккумуляторе,
            retlw       b'01001111' ; .G..DCBA = 3   и далее - возврат по стеку.
            retlw       b'01100110' ; .GF..CB. = 4
            retlw       b'01101101' ; .GF.DC.A = 5
            retlw       b'01111101' ; .GFEDC.A = 6
            retlw       b'00000111' ; .....CBA = 7
            retlw       b'01111111' ; .GFEDCBA = 8
            retlw       b'01101111' ; .GF.DCBA = 9
;*****************************************************************************
            end                     ; Конец программы.
```

Часть этой программы мы уже разобрали, а остаток "разложим на молекулы" сейчас.
Начну с "шапки".
Вы видите, что в ней дополнительно "прописаны" 2 регистра специального назначения (регистры управления защелками портов A и B: **PortA** и **PortB**) и один регистр общего назначения **Temp**, который выполняет функции многоцелевого регистра оперативной памяти ("Фигаро здесь, Фигаро там").
В него, по ходу исполнения ПП динамической индикации, будут записываться константы различного функционального предназначения.
И еще один новый "прибамбас": названия также можно присвоить и константам.
Собственно, объяснять тут особо и нечего (см. "шапку").

Переходим к рабочей части программы.
После отработки ПП **TABLE**, возврат из нее происходит на первую команду группы команд установки запятой **movwf Temp**.
В этой группе команд, можно выставить десятичную запятую в любом знакоместе линейки, что есть "привязка" к единице измерения (например, гц, Кгц, Мгц ...).
Предположим, что запятую нужно выставить в 5-м знакоместе.
Чтобы не было путаницы, обращаю Ваше внимание на то, что в данном случае, речь идет о нумерации от 0 до 7.
Если "привязаться" к "обывательской" нумерации (от 1 до 8), то это будет соответствовать 6-му знакоместу.
Анализируем то, что мы имеем после возврата из ПП **TABLE**.
После этого возврата, в зависимости от того, какое из 8-ми малых колец индикации отрабатывается, в регистре **W**, будет записано какое-либо одно из 10-ти двоичных чисел (не забывайте, что ПИК работает только с двоичными числами) результата кодировки (или перекодировки, что по сути, одно и то же).
Напоминаю, что такая разновидность двоичных чисел позволяет отобразить, в одном из активированных знакомест, один из символов цифр (от **0** до **9**).
Если речь идет о 5-м знакоместе, то на 5-м "витке" малого кольца индикации, необходимо изменить кодировку таким образом, чтобы не воздействуя на секторы 7-сегментного индикатора с названиями **A, B, C, D, E, F, G** (иначе будет индицироваться не то, что нужно), "выставить" единицу в секторе **H** ("загорание" запятой).
В интервале времени отработки остальных 7-ми "витков" малого кольца индикации, процедура установки запятой должна игнорироваться (обходиться).
То есть, имеет место быть 2 сценария исполнения программы.
Следовательно, должна быть применена команда ветвления, проверяющая состояния какого-то флага.
Речь идет о выборе одного-разъединственного числа (из группы чисел) и "привязке", к этому числу, одного из 2-х сценариев работы программы.
Первое, что приходит на ум, - задействование флага нулевого результата **Z**.
Его состояния должны анализироваться с помощью байт-ориентированных команд **btfsc** или **btfss**.
Если задействуется флаг **Z**, то есть, осуществляется "привязка" к нулевому или ненулевому результату исполнения некой команды, влияющей на состояние этого флага, то одним из результатов исполнения этой команды должен быть ноль.
При "разборках" с флагом **Z**, мы это уже проходили (команда **subwf**).
После возврата по стеку, нужно скопировать содержимое регистра **W** в какой-нибудь регистр оперативной памяти.
В данном случае, это регистр **Temp** (**movwf Temp**).
Необходимость этого сохранения обусловлена тем, что в дальнейшем, регистр **W** будет задействован, и если не принять мер по сохранению его содержимого, то оно, после записи "по верху", будет "потеряно".
Если содержимое регистра **W** сохранено в регистре **Temp**, то регистр **W** "высвобождается" и в него можно записать константу **.05** (**movlw 5**), для последующего ее использования в операции вычитания.
Для разнообразия, в команде **movlw 5**, я указал числовое значение константы без атрибутов систем исчисления (о такой возможности говорилось ранее), но можно "настучать" и **.05** или **05h**, или **b'00000101'**.
Далее, командой **bsf Status,Z** флаг **Z** поднимается.
По большому счету, эта команда не нужна (**Z** – флаг 1-й группы), но я ее ввел в текст программы для удобства отслеживания работы процедуры в симуляторе (в обучающих целях).
При этом, я исходил из того, что 5-е знакоместо одно, а количество остальных знакомест – 7.
Поэтому и поднял флаг **Z**.
Если эта команда Вам не нравится, то можете ее убрать.
После исполнения команды **subwf Index,W**, флаг **Z** будет либо опущен (если в регистре **Index** "лежит" одно из чисел **.00, .01, .02, .03, .04, .06, .07**), либо поднят (если в регистре **Index** "лежит" число **.05**).
Если флаг **Z** поднялся, то сначала исполнится "виртуальный" **NOP**, а затем будет исполнена команда **bsf Temp,7**.

В регистре **Temp**, "лежит" результат перекодировки, в котором запятая программно "погашена" (бит **№7** = **0**).
Установка единицы в бите **№ 7** (**bsf Temp,7**), соответствует активации запятой ("загорится" сектор **H** 5-го знакоместа).
Так как для установки запятой применяется бит-ориентированная команда, то она не будет воздействовать на секторы **A, B, C, D, E, F, G** (воздействует только на сектор **H**).
После исполнения команды **bsf Temp,7**, рабочая точка программы переходит на метку **No_Dot**.
Если флаг **Z** опущен, то с помощью команды **goto No_Dot**, рабочая точка программы переходит на метку **No_Dot**, "перескакивая" через команду **bsf Temp,7** (это называется ***обходом***).
В этом случае, запятая не может быть выставлена.
В конечном итоге, в 5-м знакоместе, запятая будет "высвечиваться" одновременно с "высвечиванием" символа цифры, а в остальных знакоместах, сектор **H** будет "погашен", и на индикацию будут выводиться только символы цифр.
Вопрос: "Как изменить положение запятой"?
Ответ: очень просто. В команде **movlw 5**, пятерку нужно заменить на номер того знакоместа, в котором ее нужно выставить.
Всё. Запятая выставлена.
Переходим к исполнению группы команд вывода десятичной цифры (символа) на индикацию.
Так же, как и "все пути ведут в Рим", оба сценария группы команд установки запятой "уперлись" в команду **movf Temp,W**.
Далее, **movwf PortB** → подготовка к выводу символа на индикацию.
"Подготовка" потому, что нужное знакоместо не активировано.
Значит, нужно озаботиться формированием адресного кода, который управляет внешним дешифратором.
Что и сделано: **movf Index,W** и **movwf PortA**.
После этого, символ "высветится" в нужном знакоместе, а проще говоря, он будет выведен на индикацию.
На всякий случай, "сделаю второй заход" (хуже от этого не будет).
В регистре **Temp**, находится двоичное число (результат кодировки), предназначенное для отображения в текущем знакоместе, в виде символа цифры.
После исполнения команды **movf Temp,W**, это число копируется в регистр **W** (сразу его скопировать в защелки порта B нельзя), а затем, из регистра **W**, в защелки (во все 8 штук) порта B (**movwf PortB**).
Таким образом, на анодах светодиодов линейки, выставлен код одного из символов цифр (**0 ... 9**).
Какое именно знакоместо будет активировано, зависит от того, каким будет числовое значение адресного кода, который управляет внешним дешифратором.
Числовое значение этого кода, в свою очередь, зависит от того, на каком именно, из 8-ми "витков" большого кольца индикации (одно большое кольцо индикации имеет в своем составе 8 малых колец индикации), находится рабочая точка программы.
Как раз этим "делом и занимается" группа команд формирования адресного кода.
Задача сводится к выводу, в первые 3 защелки порта A, числа (**.00** … **.07**), "лежащего" в регистре **Index** ("рулевой").
Для этого достаточно всего двух команд: **movf Index,W** и **movwf PortA**.
Естественно, что в этом случае, выводы порта A, управляющие внешним дешифратором, перед "влётом" в ПП динамической индикации, должны быть настроены на работу "на выход".
После выхода рабочей точки программы из ПП динамической индикации, их можно перестраивать под решение других задач. Но только после этого выхода.
Направления работы остальных 2-х выводов порта A не важны (они не задействуются в ПП динамической индикации), но эти выводы могут быть как-то задействованы в ***"группах подпрограмм и команд, осуществляющих различные операции"***, что на практике и происходит.
Соответственно, направления работы этих 2-х выводов будут определяться тем, что происходит в этих "различных операциях".
Идем дальше.
После того, как адресный код управления дешифратором сформирован (активировалось одно из знакомест), рабочая точка программы устанавливается на первую команду группы

команд задержки.
Эта группа команд есть не что иное, как стандартная ПП задержки без "врезок", на основе простейшего, однобайтного счетчика.
Это мы уже разбирали ранее, так что остановлюсь только на особенностях.
Команда **movlw** обращается к константе не напрямую, а через ее название, "прописанное" в "шапке" "программы".
Регистр **Temp** последний раз задействовался в группе команд вывода символов на индикацию.
При этом, свою функцию он исполнил, и на момент начала группы команд задержки, он является "свободным как ветер в поле".
А раз это так, то его можно повторно задействовать, но уже для записи в него (через регистр **W**) константы **Const1**, что и сделано.
Подобного рода многофункциональность я называю совмещением функций ("Фигаро здесь, Фигаро там").
С регистрами **Index** или **Count** подобные "манипуляции проделывать" нельзя, так как любой из них, в течение бОльшей части цикла динамической индикации, выполняет одну функцию и поэтому, "со свободой у них проблемы".
Изменяя величину константы **Const1** (это делается в "шапке"), можно "грубо" регулировать время отработки одного цикла малого кольца индикации, а соответственно, и большого тоже.
Вопрос: "Можно ли не присваивать константе названия **Const1** (а также и **Const2**), а указать ее числовое значение в команде **movlw ...**"?
Ответ: конечно же можно.
В этом случае, нужно убрать, из "шапки", соответствующие "прописки".
Идем дальше.
Следующие несколько групп команд я пропускаю по причине того, что их работа была рассмотрена на 1-м этапе.
Если необходимо сформировать калиброванный интервал времени, в состав которого входит интервал времени отработки полного цикла динамической индикации, то числовое значение константы **Const1** нужно подобрать таким образом, чтобы интервал времени отработки полного цикла динамической индикации был немного меньше требуемой величины калиброванного интервала времени (***"грубая доводка"***).
После этого, с помощью дополнительной задержки (осуществляется подбор числового значения ее константы/констант), нужно точно "довести" суммарный интервал времени отработки полного цикла динамической индикации, этой задержки и "сопутствующих" команд, до требуемой величины калиброванного интервала времени (***"точная доводка"***).
"Точная доводка" осуществляется сразу же после окончания полного цикла динамической индикации.
Посмотрите в текст программы.
По своей "конструкции", группа команд "точной доводки" - практически то же самое, что и группа команд "грубой доводки".
На момент начала "точной доводки", регистр **Temp** "свободен", и по этой причине, он в очередной раз задействуется.
Если возможны допуски, и в связи с этим, точной калибровки не требуется, то группу команд "точной доводки" можно исключить из текста программы.
После выхода рабочей точки программы из ПП динамической индикации, один из 7-сегментных индикаторов линейки будет находиться в активном состоянии, вплоть до начала следующего цикла динамической индикации.
Хотя практического (визуального) выигрыша в "гашении" этого 7-сегментного индикатора и нет, но "для порядка", его лучше погасить, установив все защелки порта B в **0**.
А можно и не "гасить". Как говорится, дело хозяйское.
В данном случае, ПП динамической индикации оформлена в виде универсальной "заготовки", "врезанной" в некую программу, составные части которой представлены в виде строк из точек.
Обратите внимание на несколько строк из точек, которые расположены после последней команды группы команд гашения активного разряда линейки.
В этом "месте" должно формироваться двоичное число, десятичное значение которого и будет индицироваться в линейке из 8-ми 7-сегментных индикаторов.
Если в линейке **8** знакомест, то необходимо сформировать **4**-байтное двоичное число

(если **7** знакомест, то **3**-байтное. Почему? Об этом говорилось ранее).
В данном, "учебно-тренировочном" случае, не имеет значения, каким именно образом сформировано 4-байтное двоичное число.
Вариантов его формирования может быть множество.
Их выбор зависит от функционального предназначения разрабатываемого устройства и выбранного способа его аппаратной и программной реализации.
Количество "рабочих" знакомест линейки можно изменить.
Если требуется количество знакомест меньшее, чем **8**, то в группе команд ***Увеличение на 1 содержимого счетчика ...***, нужно заменить число **.08** на число десятичных разрядов, которое требуется (**.07**, **.06**, ...).
Например, конструируется устройство с использованием 4-х знакомест.
Заменяем **.08** на **.04**.
После этого, в группе команд установки запятой, нужно заменить пятерку на тот номер знакоместа, в котором нужно выставить запятую и определиться с времязадающими константами **X**, **Const1** и **Const2**.
В этом случае, строки с "пропиской" регистров **LED4 ... LED7**, из "шапки" программы, нужно удалить (не используются).
Если запятую выставлять не нужно, то все команды группы команд установки запятой и первую команду группы команд вывода десятичной цифры на индикацию (вместе с названием метки **No_Dot**), можно удалить.
В этом случае, возврат (из ПП **TABLE**) будет происходить на команду **movwf PortB**.
Если 8-ми знакомест мало и их количество нужно увеличить, то принцип тот же самый, за исключением того, что в этом случае, потребуется наращивание разрядности адресного кода (управление дешифратором) с трех до четырех (нужен еще один вывод порта), замена дешифратора 555ИД7 на дешифратор (с "бегающим" нулем), имеющий 4 адресных входа и более 8-ми выходов, увеличение числа **LED**ов, корректировка "байтности" и т.д.
С этой универсальной "заготовкой" вполне можно работать, но "для полного счастья", не хватает "разборок" с ПП преобразования двоичных чисел в двоично-десятичные (ПП Bin2_10)
Об этой ПП будет рассказано в следующем разделе.

Дополнительно

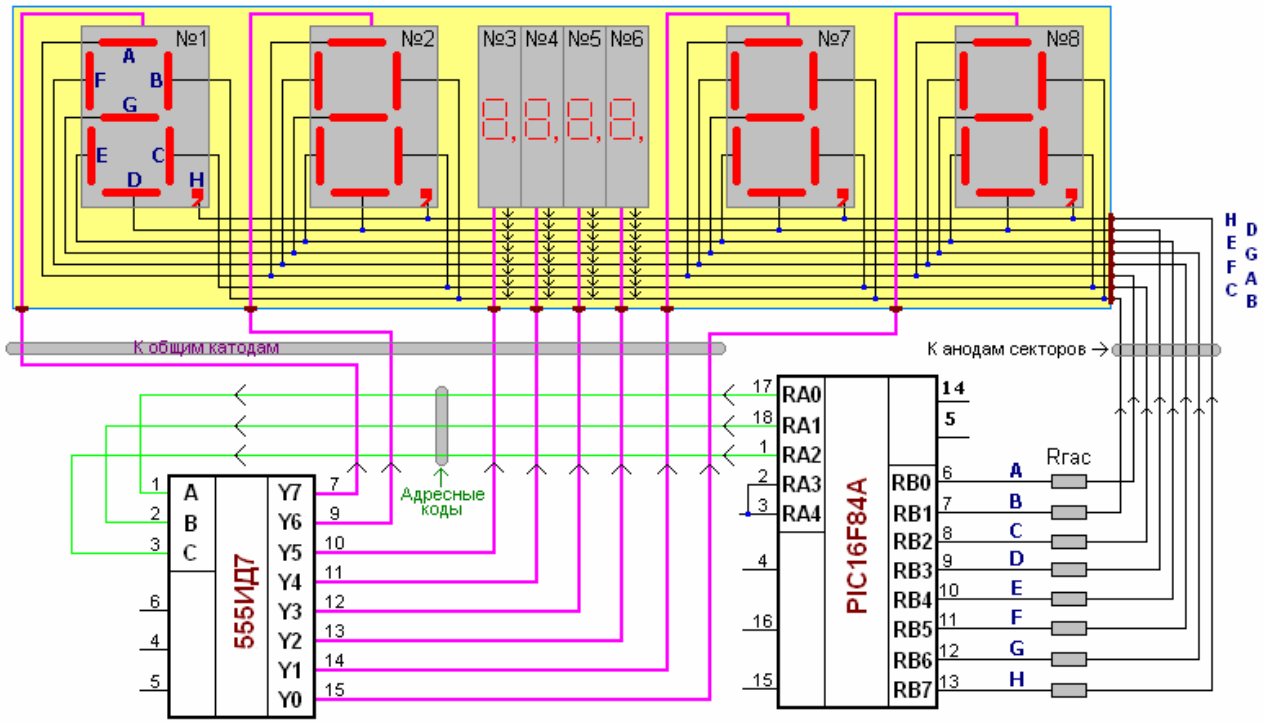
В обучающих целях, группа команд работы с содержимым регистра **Index** программы **Fsr.asm** (начинается с команды **incf Index,F**, а заканчивается командой **goto CYCLE**) предоставлена в том виде, в котором Вы видите ее в программе **Fsr.asm**.
В ней "фигурирует" восьмерка, которая "бросается в глаза". Так удобнее.
Но можно "зайти и с другого бока".
При переходе от числа **7** к числу **8**, бит **№3** изменяет свое состояние с **0** на **1**, а во всех остальных случаях (от **0** до **7** включительно), он имеет значение **0**.
Вот за это и можно "зацепиться".
Получается такая компактная группа команд:

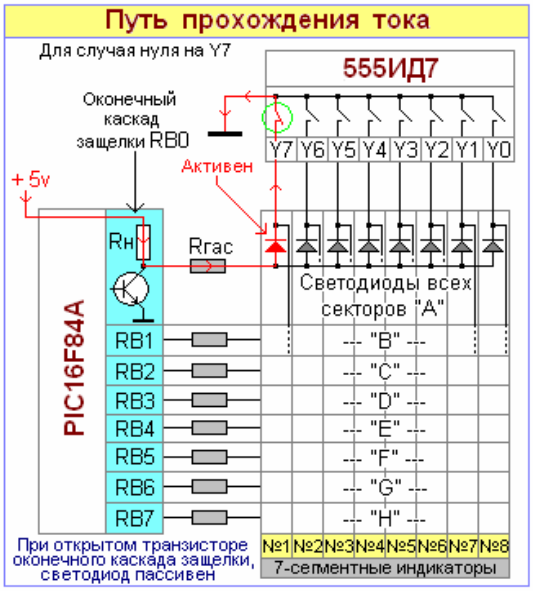
```
incf    Index,F
btfss   Index,3
goto    CYCLE
```

которой смело можно заменить соответствующую группу команд программы **Fsr.asm**.
Это называется многовариантностью.
Люди пытливого склада ума найдут в этом много интересного.

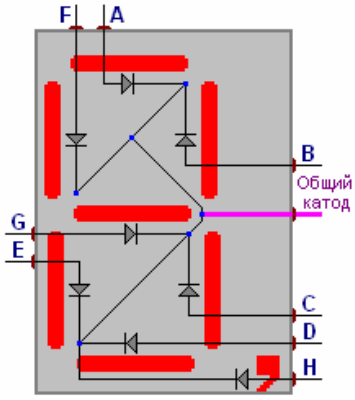
А это картинка, поясняющая принцип динамической индикации (используется линейка из восьми 7-сегментных индикаторов с общим катодом):

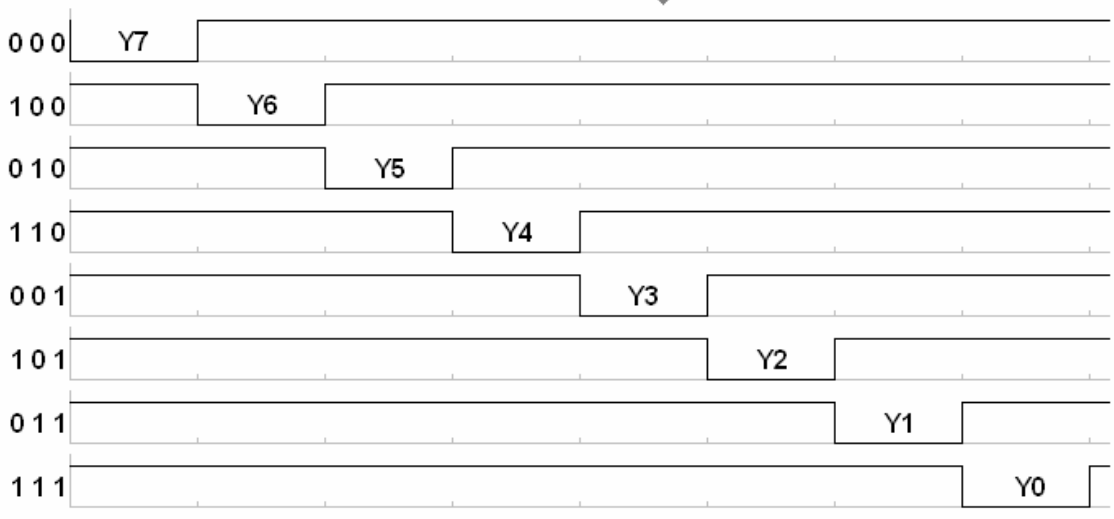
Организация 8 - разрядной динамической индикации.





Адресный код			0 на ...
A	B	C	
0	0	0	Y7
1	0	0	Y6
0	1	0	Y5
1	1	0	Y4
0	0	1	Y3
1	0	1	Y2
0	1	1	Y1
1	1	1	Y0





16. Преобразование двоичных чисел в двоично-десятичные. Окончательное формирование текста подпрограммы динамической индикации.

Работа подпрограммы преобразования двоичных чисел в двоично-десятичные достаточно сложна для понимания, и скорее всего, для начинающих, детальный "разбор этих полетов" будет связан с немалыми трудностями.
С другой стороны, "разборки" с ней, хотя бы, на уровне общих понятий, абсолютно необходимы, так как если нужно визуально отобразить числа в десятичном виде, то ***без этой подпрограммы просто не обойтись***.
Если я "выдам на гора" готовую подпрограмму, то она будет подобна "бесплатному сыру" ("расплата" наступит при первом же затруднении), а если "уйду в детали", то с непривычки, может получиться "перенапряг и отбитие почек".
То есть, необходим разумный компромисс.
Так и сделаю.
А заодно, "попутно" разберемся с некоторыми "новенькими штучками" и приобретем полезные навыки.
ПП преобразования двоичных чисел в двоично-десятичные (далее, для краткости, просто ***"ПП преобразований чисел"***) входит в состав некой условной программы **Bin2_10.asm**, в которой ПП преобразований чисел работает в комплексе с ПП динамической индикации, а обе они, в свою очередь, "органически вписаны" в программу, реализующую некое измерительное устройство.
Результаты измерений выводятся на индикацию в линейку из 8-ми 7-сегментных индикаторов.
По большому счету, в данном случае, неважно, что именно измеряется.
Главное - чтобы на каждом "витке" полного цикла "программы", к моменту "влёта" рабочей точки программы в ПП преобразований чисел, тем или иным способом, было бы сформировано 4-байтное двоичное число.
С ним и будет работать ПП преобразований чисел, а после этого, ПП динамической индикации.

Файл программы называется **Bin2_10.asm** (находится в папке ***"Тексты программ"***).

Она выглядит так:

```
;*******************************************************************************
;          ГРУППА ПОДРПРОГРАММ ПРЕОБРАЗОВАНИЯ ДВОИЧНЫХ ЧИСЕЛ
;                          В ДВОИЧНО-ДЕСЯТИЧНЫЕ
; (для случая преобразования 4-байтных двоичных чисел в 8-разрядные  десятичные).
;*******************************************************************************
;                                "ШАПКА ПРОГРАММЫ"
;*******************************************************************************
; Bin2_10.asm   Универсальная группа подпрограмм преобразования 4-байтных
;                     двоичных чисел в 8-разрядные десятичные числа.
;.....................................................
;==============================================================================
; Определение положения регистров специального назначения.
;==============================================================================
Indf        equ         00h         ; Регистр Indf.
Status      equ         03h         ; Регистр Status.
FSR         equ         04h         ; Регистр FSR.
;.....................................................
;.....................................................
;==============================================================================
; Определение названия и положения регистров общего назначения.
;==============================================================================
LED0        equ         10h         ; Регистр хранения результатов преобразований
                                    ; 1-го двоично-десятичного разряда.
LED1        equ         11h         ; ------- 2-го ------------------------
LED2        equ         12h         ; ------- 3-го ------------------------
LED3        equ         13h         ; ------- 4-го ------------------------
LED4        equ         14h         ; ------- 5-го ------------------------
```

```
LED5        equ         15h         ; ------- 6-го ------------------------
LED6        equ         16h         ; ------- 7-го ------------------------
LED7        equ         17h         ; ------- 8-го ------------------------

Count       equ         0Dh         ; Счетчик проходов.
Mem         equ         1Fh         ; Регистр оперативной памяти.
TimerL      equ         1Bh         ; Регистр младшего разряда 4-байтного
                                    ; двоичного числа.
TimerM      equ         1Ch         ; Регистр среднего разряда 4-байтного
                                    ; двоичного числа.
TimerH      equ         1Dh         ; Регистр старшего разряда 4-байтного
                                    ; двоичного числа.
TimerHH     equ         1Eh         ; Регистр самого старшего разряда 4-байтного
                                    ; двоичного числа.
;.....................................................
;.....................................................
;=====================================================================
; Определение места размещения результатов операций.
;=====================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;=====================================================================
; Присваивание битам названий.
;=====================================================================
C           equ         0           ; Флаг переноса-заёма.
;.....................................................
;.....................................................
;=====================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*********************************************************************

;*********************************************************************
;                          РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;*********************************************************************
START       ....................................
;           ....................................
NEW         call        Bin2_10     ; Условный переход в ПП Bin2_10
                                    ; Адрес следующей команды закладывается в стек
;           ....................................
;           ....................................
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; ГРУППА ПОДПРОГРАММ 8-РАЗРЯДНОЙ ДИНАМИЧЕСКОЙ ИНДИКАЦИИ без ПП TABLE
; (то, что было рассмотрено ранее).
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;....        ....................................
;           ....................................
;....        ....................................
;           ....................................
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; ГРУППА ПОДПРОГРАММ ФОРМИРОВАНИЯ 4-БАЙТНОГО ДВОИЧНОГО ЧИСЛА.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;....        ....................................
;           ....................................
;....        ....................................
;           ....................................
;           ....................................
; К этому моменту 4-байтное двоичное число (в регистре
; TimerL/TimerM/TimerH/TimerHH) должно быть сформировано для последующей
; обработки его в группе подпрограмм преобразования 4-байтных двоичных чисел
; в 8-разрядные десятичные числа.

            goto        NEW         ; Безусловный переход на метку NEW
                                    ; подпрограммы START, то есть, начало нового
                                    ; полного цикла "программы".
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
; ГРУППА ПОДПРОГРАММ ПРЕОБРАЗОВАНИЯ 4-БАЙТНЫХ ДВОИЧНЫХ ЧИСЕЛ В 8-РАЗРЯДНЫЕ
; ДЕСЯТИЧНЫЕ ЧИСЛА.
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Подготовка к преобразованию.
;==============================================================================
Bin2_10     bcf         Status,C    ; Сброс флага переноса-заёма.
            movlw       .32         ; Запись в регистр Count числа проходов
            movwf       Count       ; преобразования, равного суммарному
                                    ; количеству битов многоразрядного регистра
                                    ; TimerL/TimerM/TimerH/TimerHH (8*4=32).
            clrf        LED0        ; Сброс в 0 содержимого регистра LED0.
            clrf        LED1        ; -------"------ LED1.
            clrf        LED2        ; -------"------ LED2.
            clrf        LED3        ; -------"------ LED3.
            clrf        LED4        ; -------"------ LED4.
            clrf        LED5        ; -------"------ LED5.
            clrf        LED6        ; -------"------ LED6.
            clrf        LED7        ; -------"------ LED7.
;------------------------------------------------------------------------------
; Примечание: процесс преобразования заканчивается при уменьшении числа проходов
;             преобразования, которые заложены в регистр Count (.32), до нуля.
;==============================================================================
; Циклический сдвиг влево.
;==============================================================================
Loop16      rlf         TimerL,F    ; Циклический сдвиг влево 4-байтного двоичного
            rlf         TimerM,F    ; числа, записанного в группе регистров
            rlf         TimerH,F    ; TimerL/TimerM/TimerH/TimerHH, на одну
            rlf         TimerHH,F   ; позицию через бит C регистра STATUS.

            rlf         LED0,F      ; Циклический сдвиг влево 4-байтного двоичного
            rlf         LED1,F      ; числа, записанного в группе регистров
            rlf         LED2,F      ; LED0/LED1/LED2/LED3, на одну позицию через
            rlf         LED3,F      ; бит C регистра STATUS.

            decfsz      Count,F     ; Декремент (-1) содержимого регистра Count с
                                    ; сохранением результата в нем же.
            goto        adjDEC      ; Если результат не=0, то переход в ПП adjDEC
                                    ; Если  результат =0, то программа
                                    ; исполняется далее.
;==============================================================================
; Поразрядное распределение содержимого регистров LED0...3 (обоих
; полубайтов) по младшим полубайтам регистров LED0...7.
;==============================================================================
            swapf       LED3,W      ; Запись старшего полубайта LED3
            andlw       0Fh         ; в младший полубайт LED7.
            movwf       LED7        ; --------------------------------

            movfw       LED3        ; Запись младшего полубайта LED3
            andlw       0Fh         ; в младший полубайт LED6.
            movwf       LED6        ; --------------------------------

            swapf       LED2,W      ; Запись старшего полубайта LED2
            andlw       0Fh         ; в младший полубайт LED5.
            movwf       LED5        ; --------------------------------

            movfw       LED2        ; Запись младшего полубайта LED2
            andlw       0Fh         ; в младший полубайт LED4.
            movwf       LED4        ; --------------------------------

            swapf       LED1,W      ; Запись старшего полубайта LED1
            andlw       0Fh         ; в младший полубайт LED3.
            movwf       LED3        ; --------------------------------

            movfw       LED1        ; Запись младшего полубайта LED1
            andlw       0Fh         ; в младший полубайт LED2.
            movwf       LED2        ; --------------------------------
```

```
            swapf       LED0,W      ; Запись старшего полубайта LED0
            andlw       0Fh         ; в младший полубайт LED1.
            movwf       LED1        ; -------------------------------

            movfw       LED0        ; Запись младшего полубайта LED0
            andlw       0Fh         ; в младший полубайт LED0.
            movwf       LED0        ; -------------------------------
;-----------------------------------------------------------------
; Конец распределения. В младших полубайтах регистров LED0...7
; установлены двоично-десятичные числа в порядке возрастания разрядности.
; Старшие полубайты = 0.
;-----------------------------------------------------------------
            return                  ; Переход по стеку в группу подпрограмм
                                    ; 8-разрядной динамической индикации.
;=================================================================================
; Запись в регистр FSR адресов регистров LED0...3 для дальнейшей косвенной
; адресации к ним в ПП adjBCD.
; Переход к обработке следующего LED - после возврата по стеку.
;=================================================================================
adjDEC      movlw       LED0        ; Запись в регистр FSR, через регистр W,
            movwf       FSR         ; адреса регистра LED0 с дальнейшим переходом
            call        adjBCD      ; в ПП adjBCD (адрес следующей команды
                                    ; закладывается в стек).
;---> Возврат по стеку из ПП adjBCD.
            movlw       LED1        ; ----------------------------
            movwf       FSR         ; То же самое для регистра LED1.
            call        adjBCD      ; ----------------------------
;---> Возврат по стеку из ПП adjBCD.
            movlw       LED2        ; ----------------------------
            movwf       FSR         ; То же самое для регистра LED2.
            call        adjBCD      ; ----------------------------
;---> Возврат по стеку из ПП adjBCD.
            movlw       LED3        ; ----------------------------
            movwf       FSR         ; То же самое для регистра LED3.
            call        adjBCD      ; ----------------------------
;---> Возврат по стеку из ПП adjBCD.
            goto        Loop16      ; Проход всех LED (с LED0 по LED3). Переход в
                                    ; ПП Loop16, то есть на следующее кольцо
                                    ; числовых преобразований.
;=================================================================================
; Основные операции преобразования двоичных чисел в двоично-десятичные:
; операции сложения LED0...3 и констант 03h,30h с условиями по 3-му и 7-му битам.
;=================================================================================
adjBCD      movlw       3           ; Сложить содержимое текущего LED (LED0...3) с
            addwf       0,W         ; числом 03h, с записью результата операции,
            movwf       Mem         ; через регистр W, в регистр Mem.

            btfsc       Mem,3       ; Анализ состояния 3-го бита регистра Mem.
            movwf       0           ; Если бит № 3 =1, то содержимое регистра Mem
                                    ; копируется в текущий LED.
            movlw       30          ; Если бит №3 =0, то содержимое текущего LED
            addwf       0,W         ; складывается с константой 30h, с последующей
            movwf       Mem         ; записью результата операции, через регистр
                                    ; W, в регистр Mem.
            btfsc       Mem,7       ; Анализ состояния 7-го бита регистра Mem.
            movwf       0           ; Если бит №7 =1, то содержимое регистра Mem
                                    ; копируется в текущий LED.
            retlw       0           ; Если бит №7 =0, то регистр W очищается и
                                    ; происходит возврат по стеку в ПП adjDEC.
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; ГРУППА КОМАНД ПРЕОБРАЗОВАНИЯ ДВОИЧНО-ДЕСЯТИЧНОГО КОДА В КОД 7-СЕГМЕНТНОГО
;     ИНДИКАТОРА (относится к группе подпрограмм динамической индикации).
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
TABLE       ....................................
;           ....................................
```

```
;               .....................................
;*****************************************************************************
                end                             ; Конец программы.
```

Сначала разберемся с общей "конструкцией" программы.
В начале ее исполнения, рабочая точка программы начинает свое движение по "линейному" участку ПП **START**, начиная с 1-й ее команды и до команды **call Bin2_10**.
После исполнения команды **call Bin2_10**, рабочая точка программы "прыгает" на 1-ю команду ПП преобразований чисел **Bin2_10**, после чего она начнет исполняться.
В результате исполнения команды **call Bin2_10**, адрес команды, следующей за командой **call Bin2_10**, "закладывается" в стек и находится там до окончания отработки ПП **Bin2_10**.
После окончания отработки ПП **Bin2_10** (после исполнения команды **return**), из вершины стека "выгружается" указанный выше адрес, и после отработки всех оставшихся команд ПП **START**, начинается исполнение ПП динамической индикации (то, о чем говорилось в предыдущем разделе).
После отработки ПП динамической индикации, рабочая точка программы переходит в ПП формирования 4-байтного, двоичного числа.
Обращаю Ваше внимание на то, что этот переход происходит без использования команды перехода, так как работа происходит на "линейном" участке программы (просто исполняется следующая команда).
"Конструкций" подпрограмм формирования 4-байтного, двоичного числа может быть множество. Это зависит от функциональности устройства, которое обслуживает программа. Главное, чтобы в результате работы этой ПП, тем или иным способом, было сформировано 4-байтное, двоичное число.
После того как это произошло, для обеспечения перехода в ПП преобразований чисел ("закольцовки"), осуществляется безусловный переход на метку **NEW**, входящую в состав ПП **START**.
Таким образом, мы "прошлись" по кольцу полного цикла программы.
По ходу исполнения программы, таких колец может быть "намотано" множество.
Это зависит от времени, в течение которого устройство находится во включенном состоянии.
Разбираемся с возможными "непонятками".
Вопрос: "Зачем осуществлять безусловный переход на метку **NEW**? Если этого не делать (команда **goto NEW** и метка **NEW** отсутствуют), то из ПП формирования 4-байтного, двоичного числа, в ПП преобразований чисел, рабочая точка программы перейдет самоходом (стык находится на линейном участке программы)".
Ответ: да, таким образом рабочая точка программы войдет в ПП преобразований чисел, но выйти из нее она не сможет, так как стек будет пустой (куда переходить? "Глюк").
А пустой он будет потому, что перед исполнением ПП преобразований чисел, в него не "заложился" адрес возврата (не исполнена команда **call**).
Для того чтобы обеспечить эту "закладку", а заодно и вернуться "туда, куда положено", необходимо выполнить команду **goto NEW** (перейти на исполнение команды **call Bin2_10**).
Разбираемся с ПП **START**.
С точки зрения обеспечения надежности работы устройства, полный цикл программы лучше начинать с исполнения ПП **START**, а не с команды, которая не входит в ее состав.
В противном случае, ПП **START** будет исполнена всего один раз (после включения питания устройства, на 1-м "витке" полного цикла программы), и в ходе дальнейшей работы программы (от 2-го "витка" и далее), ПП **START** будет обходиться.
В этом случае, при неконтролируемом изменении содержимого задействованных в программе регистров (например, в результате какого-то сбоя), существует вероятность "зависания" программы.
Если предприняты меры по "борьбе с зависаниями" (например, работает сторожевой таймер), то в принципе, это не страшно, но если этих мер не предпринято или существуют сомнения, или речь идет об обеспечении максимальной надежности работы устройства, то перестраховаться не помешает.
При наличии такой перестраховки, последствия некоторых сбоев будут ликвидированы за счет приведения "поврежденного" содержимого регистров к норме.
Насколько я понимаю, в ответах на этот "скользкий вопрос", единого мнения нет.
Можно, начиная со 2-го "витка" полного цикла программы, обойти ПП **START**.
Можно не обходить.

Можно включить в полный цикл программы (от 2-го "витка" и далее) наиболее "ответственные", подготовительные операции ПП **START** (это что-то типа компромиссного варианта).
Реализацию именно такого подхода Вы и видите в тексте программы **Bin2_10.asm**.
"Ответственные" (эта оценка субъективна) команды подготовительных операций ПП **START** располагаются ниже команды **call Bin2_10**, а команды подготовительных операций, "рангом пониже", располагаются выше ее.
Это означает то, что команды подготовительных операций, располагающиеся выше команды **call Bin2_10**, будут исполнены на 1-м "витке" полного цикла программы, и в дальнейшем (от 2-го "витка" и далее), они исполняться не будут.
Команды подготовительных операций, располагающиеся ниже команды **call Bin2_10**, будут исполняться на каждом "витке" полного цикла программы.
В "перестраховочном" случае, нужно заменить команду **goto NEW** на команду **goto START**.
Если есть такое желание, то ничто не мешает осуществить такую замену.
А теперь давайте разберемся в сути преобразований двоичных чисел в двоично-десятичные.
Предположим, что группа ПП формирования 4-байтного, двоичного числа "выдала на гора" 4-байтное, двоичное число.
По определению, такое число должно быть "заложено" в 4 регистра общего назначения.
Пусть они будут называться **Timer...**
Так как регистров 4 штуки, то нужно определиться с порядком их старшинства:

TimerHH, TimerH, TimerM, TimerL.

Ранее об этом говорилось.
Если нужно что-то подсчитать, то речь идет о многоразрядном, двоичном счетчике, которому можно поставить в соответствие, например, двоичный счетчик на 8-ми микросхемах 555ИЕ5 (каждая 555ИЕ5 работает с полубайтом, а не с байтом, и поэтому таких м/схем нужно 8 штук).
Эта аналогия относится к форме представления результатов счета.
По этому показателю, аналогия полная.
То есть, в обоих случаях, 32-битное, двоичное число будет отображаться в стандартном весовом коде.
Что же касается принципов счета, то существуют весьма существенные различия, связанные с тем, что в 555ИЕ5, счет реализуется только аппаратными средствами.
Пошли дальше.
Линейка 7-сегментных индикаторов, состоящая из 8-ми знакомест, может отобразить десятичное число не более **99 999 999**.
Двоичный эквивалент этого числа в стандартном весовом коде:

00000101 11110101 11100000 11111111

См. конвертор систем исчисления, который у Вас имеется (можете проверить).
Вопрос: "Что такое двоично-десятичное число"?.
Ответ: ***двоично-десятичное число это одно из чисел числового диапазона 0 … 9, представленное в двоичной форме.***
Двоично-десятичное число, по определению, не может быть больше **9**-ти.
Из этого следует то, что оно отображается только в младшем полубайте байта, а старший полубайт байта всегда будет равен нулю (**0000xxxx**).
Например, десятичное число **99 999 999**, в двоичном виде, может быть отражено в 4-байтном регистре, но в двоично-десятичном виде, такое число не может быть отражено в четырех регистрах.
В последнем случае, для отображения этого числа потребуется 8 регистров.
Число **99 999 999**, в двоично-десятичном виде, выглядит так:

00001001 00001001 00001001 00001001 00001001 00001001 00001001 00001001

Вы видите, что во всех младших полубайтах всех 8-ми регистров, "лежат" девятки, а во всех старших полубайтах, нули.
Вот зачем и нужны 8 регистров **LED0...7**, "прописку" которых Вы видите в "шапке" программы.
Первые четыре из них (**LED0...3**), в ПП преобразований чисел, задействуются для хранения промежуточных и конечных результатов преобразований.
Регистры **LED4...7** для этого не задействуются.
На конечной стадии работы ПП преобразований чисел, из младших и старших полубайтов

регистров **LED0...3**, в младшие полубайты регистров **LED0...7** (это не ошибка, никакой ошибки нет), переписываются конечные результаты преобразований чисел.
Это и есть то, что в дальнейшем, подлежит кодировке.
При этом, в старших полубайтах **LED0...7** "выставляются" нули.
Это как раз то, что Вы видите в примере (для случая 99999999).
В конечном итоге, в младших полубайтах регистров **LED0...7**, на все время полного цикла ПП динамической индикации, фиксируются двоично-десятичные числа, с которыми эта ПП и будет работать.
В общем виде, работу ПП преобразований чисел можно описать так:
После "влёта" рабочей точки программы в ПП преобразований, в предварительно назначенный счетчик "проходов" **Count**, записывается константа, значение которой равно суммарному количеству битов 4-байтного регистра **TimerHH/TimerH/TimerM/TimerL** (в нем находится 4-байтное, двоичное число, которое нужно преобразовать в двоично-десятичное).
То есть, в данном случае, числовое значение этой константы, должно быть равно **.32**.
На каждом из 32-х "витков" внутреннего цикла ПП преобразований чисел, содержимое счетчика "проходов" **Count** декрементируется, и если результат декремента не равен **0**, то происходит переход рабочей точки программы на следующий "виток" внутреннего цикла ПП преобразований чисел.
Таким образом, войдя в ПП преобразований чисел, рабочая точка программы "наматывает" 32 "витка" внутреннего цикла ПП преобразований чисел, которые, в сумме, составляют полный цикл ПП преобразований чисел.
После этого, она выходит из ПП преобразований чисел по сценарию "программа исполняется далее".
При этом, содержимое счетчика "проходов" **Count** равно нулю.
Что происходит, в процессе "прохождения" рабочей точкой программы, каждого из 32-х "витков" внутреннего цикла ПП преобразований чисел?
Сначала производится групповой, циклический сдвиг содержимого 8-байтного регистра **TimerL/TimerM/TimerH/TimerHH/LED0/LED1/LED2/LED3** влево (через бит **C** регистра **STATUS**).
Перед этим, (в начале исполнения полного цикла ПП преобразований чисел), в регистры **LED0...7** записываются нули.
То есть, их "старое" содержимое удаляется, после чего они готовы к работе.
В результате каждого такого группового сдвига (последовательный сдвиг содержимого 8-ми регистров. 8 команд **RLF**), происходит перенос содержимого старшего бита регистра **TimerHH** в младший бит регистра **LED0**.
Таким образом, через 32 таких сдвига, содержимое **TimerL/TimerM/TimerH/TimerHH** переместится в **LED0/LED1/LED2/LED3**.
Если это трудно себе представить, то вспомните про "бегущую строку" или "бегущие огни".
Если просто переместить содержимое **TimerL/TimerM/TimerH/TimerHH** в **LED0/LED1/LED2/LED3**, то это ничего полезного не даст, так как конечным результатом такого перемещения будет исходное, двоичное число.
Следовательно, после очередного группового сдвига, необходимо преобразовать содержимое 4-байтного регистра **LED0/LED1/LED2/LED3** таким образом, чтобы на момент завершения всех 32-х групповых сдвигов (счетчик проходов **Count** очищен), во всех 8-ми полубайтах 4-байтного регистра **LED0/LED1/LED2/LED3**, с соблюдением порядка старшинства, "осели" **двоично-десятичные числа** (любое из них можно отобразить в полубайте), точно отражающие значение исходного, **двоичного числа**.
То есть, речь идет о процедуре типа "чтение – модификация – запись.
Так как запись результатов числовых преобразований производится в те же регистры, из которых производилось чтение (**LED0...3**), то при модификации, должен задействоваться регистр оперативной памяти (в нашем случае - **Mem**), в котором будут храниться как промежуточные, так и конечные результаты преобразований чисел.
По окончании текущего цикла модификации, модифицированное число, "осевшее" в регистре **Mem**, копируется в текущий **LED**.
Предположим, что это происходит.
После того как будет "отмотано" 32 "витка" внутреннего цикла ПП преобразований чисел, во всех 8-ми полубайтах 4-байтного регистра **LED0/LED1/LED2/LED3** будут "лежать" 8 двоично-десятичных чисел.

Остается только "раскидать" их (также с соблюдением порядка старшинства) по младшим полубайтам регистров **LED0, LED1, LED2, LED3, LED4, LED5, LED6, LED7**, а в их старшие полубайты, "принудительно" записать нули (см. пример двоично-десятичного представления числа **99999999**).
Всё. Конец преобразования.
После этого, содержимое регистров **LED0...7** можно обрабатывать в ПП динамической индикации.
Не смотря на достаточно сложный алгоритм числовых преобразований, работать с ПП преобразований чисел совсем не сложно.
В "шапке" программы, "прописываются" 2 регистра общего назначения: счетчик числа проходов и регистр оперативной памяти (в нашем случае, **Count** и **Mem**).
В начале ПП преобразований чисел, в счетчик проходов, нужно записать число битов двоичного числа.
Например, если нужно отобразить символы десятичных чисел не в 8-ми, а в 4-х знакоместах (максимальное значение **9999**), то речь идет о 2-байтном регистре
(например, **TimerM/TimerL**).
Таким образом, в счетчик числа проходов, необходимо записать константу 8x2=**.16**.
В этом случае, лишние **LED**ы (**LED4...7**) и группы команд, которые работают с их содержимым, нужно убрать.
Должны остаться только регистры **LED0, LED1, LED2, LED3** и те группы команд, которые "работают" с их содержимым.
Соответственно, в ПП преобразований, вплоть до ее концовки (до процедуры распределения полубайтов), нужно использовать только регистры **LED0** и **LED1**.
То есть, в этом случае, из текста программы **Bin2_10.asm**, нужно "изъять" некоторые команды.
Для того чтобы стало понятно, какие именно команды нужно "изымать" или добавлять (в случае работы с двоичными числами, отображаемыми более чем в 4-х байтах), нужно поподробнее разобраться со стратегией работы ПП преобразований чисел (**Bin2_10**).
ПП **Bin2_10** начинается с подготовительных операций.
Флаг переноса-заёма **C** опускается (бит **C** регистра **STATUS** сбрасывается в **0**), в счетчик проходов **Count** записывается константа, числовое значение которой равно общему количеству битов 4-байтного регистра **TimerL/TimerM/TimerH/TimerHH** (**.32**), и содержимое всех **LED**ов сбрасывается в **0**.
Подготовительные операции закончены.
Далее, начинается отработка ПП **Loop16**.
В ее начале, происходит циклический сдвиг влево, на одну позицию (через бит **C** регистра **STATUS**), содержимого 8-разрядного регистра
TimerL/TimerM/TimerH/TimerHH/LED0/LED1/LED2/LED3 (8 команд **rlf**),
а потом, содержимое счетчика количества проходов (**Count**) декрементируется (**decfsz Count,F**, результат декремента сохраняется в нем же).
Так как байт-ориентированная команда **decfsz** является командой ветвления, то возможны два сценария работы программы.
Предположим, что результат декремента равен **0** ("отмотаны" все 32 "витка").
В этом случае, начнется отработка группы команд поразрядного распределения содержимого регистров **LED0...3** (обеих полубайтов), по младшим полубайтам регистров **LED0...7** (концовка).
Этой группе команд вполне можно было бы "присвоить статус" отдельной подпрограммы, но так как на 1-ю команду этой группы команд, переходов нет, то формально, она входит в состав ПП **Loop16**.
Если результат декремента содержимого регистра **Count** не равен **0**, то осуществляется безусловный переход в ПП **adjDEC**.
В состав ПП **adjDEC** входят 4 группы команд, в каждой из которых осуществляется условный переход в ПП **adjBCD**.
Возврат происходит на следующую (из этих четырех) группу команд.
Каждая из этих групп команд работает со "своим" регистром **LED** (с нулевого по третий), то есть, соблюдается порядок старшинства (в начале ПП **adjDEC**, происходит "работа" с содержимым регистра **LED0**, а в конце, с содержимым регистра **LED3**).
Предположим, что с содержимым регистра **LED3** осуществлены необходимые действия и произошел возврат на команду **goto Loop16**.

После исполнения этой команды, начнется "отмотка нового витка" внутреннего цикла ПП **Bin2_10** (напоминаю, что таких "витков" 32 штуки).
На каждом таком "витке", происходит сдвиг содержимого 8-байтного регистра **TimerL/TimerM/TimerH/TimerHH/LED0/LED1/LED2/LED3** на одну позицию (байт), по направлению от **TimerL** к **LED3**.
То есть, 4-байтный регистр **LED0/LED1/LED2/LED3** будет последовательно заполняться результатами числовых преобразований.
ПП **adjDEC** никаких числовых преобразований не осуществляет.
Она исполняет функцию "администратора", то есть, определяет порядок обработки содержимого регистров **LED0...3**.
Он следующий: **LED0, LED1, LED2, LED3** (учтен порядок старшинства).
В состав каждой из этих 4-х групп команд, входят две команды процедуры "разорванной", косвенной адресации.
Слово "разорванной" означает то, что в ПП **adjDEC**, осуществляется только запись, в регистр **FSR**, адреса регистра, с содержимым которого будут производиться действия, а сами эти действия производятся в другой подпрограмме (в ПП **adjBCD**).
После исполнения команды **movlw LED...** (**0, 1, 2, 3**), в регистр **W**, записывается адрес текущего **LED**а (вспомните про то, о чем говорилось ранее).
После исполнения команды **movwf FSR**, этот адрес, из регистра **W**, копируется в регистр **FSR**
Далее, осуществляется условный переход в ПП **adjBCD** (**call adjBCD**).
После того, как ПП **adjBCD** будет отработана, то есть, будет исполнена команда **retlw 0**, содержимое регистра **W** сбрасывается в **0** и происходит возврат на команду записи, в регистр **W**, адреса следующего **LED**а.
Проще говоря, начинается обработка содержимого следующего **LED**а.
И так будет происходить до тех пор, пока не будет обработано содержимое регистра **LED3**.
После этого, осуществляется безусловный переход в ПП **Loop16**, то есть, на новый "виток" внутреннего цикла ПП **Bin2_10**.
А теперь обратим внимание на стек.
Ранее рассматривались случаи, когда в стек "закладывался" только один адрес возврата.
В данном случае, адрес возврата на 1-ю, после команды **call Bin2_10**, команду программы, будет "лежать" в стеке вплоть до окончания отработки ПП **Bin2_10**, но только преимущественно (основную часть времени), не в вершине стека (в 1-й его строке), а во 2-й строке, так как после исполнения команд **call adjBCD**, вершину стека поочередно будут занимать соответствующие адреса возвратов.
Таким образом, ***если на момент исполнения текущей команды call, в стеке уже находится адрес*** (адреса) ***возврата, "заложенный" в него ранее, то в таблице стека*** (всего в этой таблице 8 строк)***, он*** (они) ***сместится на одну позицию вниз, а в вершину стека запишется адрес возврата, соответствующий текущей*** (последней) ***команде call.***
После освобождения вершины стека, все "имеющиеся в наличии", активные адреса возвратов, синхронно сместятся вверх на одну позицию (строку)***, после чего, в вершине стека будет "лежать" адрес возврата, ранее "дислоцировавшийся" во 2-й строке.***
Итак, после записи, в регистр **FSR**, адреса текущего **LED**а (в ПП **adjDEC**), происходит условный переход в ПП **adjBCD**.
ПП **adjBCD** является как бы "основной кухней" преобразования чисел.
Она обрабатывает результаты групповых, циклических сдвигов, находящиеся в регистрах **LED0...3**.
В начале ПП **adjBCD**, с помощью команды **movlw 3**, в регистр **W**, записывается константа **03h** (**00000011**), которая далее, с помощью команды **addwf 0,W**, суммируется с содержимым текущего **LED**а.
Этой командой завершается, ранее начатая, процедура косвенной адресации.
Обращение к регистру **INDF** происходит не непосредственно (с "пропиской" его названия), а через его адрес (косвенно).
В данном случае, происходит обращение к числу **0**, то есть, к адресу регистра.
В области оперативной памяти, по адресу **00h**, "дислоцируется" регистр **INDF**.
Следовательно, обращение происходит именно к нему.

А если происходит обращение к регистру **INDF** (прямое или косвенное, без разницы), то по определению, происходит обращение к содержимому того регистра, адрес которого записан в регистре **FSR**.
В регистре **FSR**, "лежит" адрес текущего **LED**а, следовательно, произойдет суммирование содержимого текущего **LED**а с ранее записанной, в регистр **W**, константой **03h**.
Результат суммирования сохранится в регистре **W**.
Затем, результат суммирования, из регистра **W**, копируется в регистр оперативной памяти **Mem**, после чего, состояние бита **№3** анализируется (**btfsc Mem,3**).
Если в этом бите **1**, то исполняется команда **movwf 0**, которая также относится к "разряду хитрых".
В ней также (см. выше) происходит обращение не к названию регистра **INDF**, а к адресу регистра **INDF** (**00h**).
То есть, происходит обращение к содержимому текущего **LED**а.
Таким образом, команда **movwf 0** копирует содержимое регистра **W**, в текущий **LED**.
А так как в регистре **W** находится то же самое, что и в регистре **Mem**, то фактически, происходит копирование содержимого регистра **Mem**, в текущий **LED**.
Если бит **№3** регистра **Mem** равен **0**, то процедура суммирования повторится снова, но только в регистр **W** будет записана не константа **03h**, а константа **30h** (**00110000**), и будет произведен анализ состояния не бита **№3**, а бита **№7** регистра **Mem**.
Напоминаю, что если в тексте программы указано число без атрибутов систем исчисления (а в данном случае так оно и есть), то речь идет о 16-ричной системе исчисления, а не о десятичной.
Вместо **movlw 30**, можно написать **movlw .48** или **movlw b'00110000'**.
Результат будет одним и тем же.
Если бит **№7** регистра **Mem** будет равен **0**, то произойдет возврат в ПП **adjDEC**, после чего произойдет смена текущего **LED**а, следующий переход в ПП **adjBCD**, ее отработка, возврат в ПП **adjDEC** и т.д.
До тех пор, пока не будет закончена обработка содержимого регистра **LED3**.
Далее, исполняется команда **goto Loop16**, после чего начинается "отмотка" следующего витка/кольца ПП **Bin2_10**.
И так → 32 раза.
После того, как содержимое счетчика количества проходов **Count** станет равным **0**, начнется отработка группы команд поразрядного распределения содержимого регистров **LED0...3** по младшим полубайтам регистров **LED0...7** ("концовочная" процедура).
На момент начала ее отработки, в 8-ми **полу**байтах регистров **LED0...3**, в порядке старшинства, будут "лежать" результаты преобразований, в виде восьми 4-битных, двоично-десятичных чисел.
Направление возрастания старшинства: от младшего полубайта регистра **LED0**, и далее по-порядку.
Распределение начинается со старшего полубайта регистра **LED3** (он "уходит" в младший полубайт регистра **LED7**. В его старший полубайт записывается **0**) и далее, по-порядку (в сторону уменьшения разрядности) и аналогично.
Примечание: если начать это распределение с младшего полубайта регистра **LED0** ("с другого конца"), то содержимое некоторых полубайтов будет утеряно.
То есть, в данном случае (8 знакомест), в наличии должно быть 8 групп команд распределения.
Причем, при распределении каждого полубайта, должна гарантированно обеспечиваться запись нуля в старший полубайт регистра-получателя (почему? См. выше).
Эти 8 групп команд, по сути, выполняют одну и ту же "работу", но разными способами:
- 4 группы команд, работающие с младшими полубайтами регистров **LED0...3**, - одним способом,
- и 4 группы команд, работающие со старшими полубайтами регистров **LED0...3**, - другим.

Это обусловлено положением полубайта в байте.
Если полубайт старший, то необходимо задействовать команду смены местами старшего и младшего полубайтов (**swapf**), а если полубайт младший, то этого не требуется.
Рассмотрим работу первых двух групп команд (остальные пары групп команд работают аналогично).
Распределение начинается со старшего полубайта регистра **LED3**.

С помощью команды **swapf LED3,W**, старший и младший полубайты регистра **LED3** меняются местами. Результат этой операции сохраняется в регистре **W**.
С помощью команды **andlw 0Fh**, выполняется логическая операция **побитного И** содержимого регистра **W** и константы **0Fh** (**.15** или **00001111**).
То есть, фактически, выполняется **побитное И** содержимого регистра **LED3** (с учетом смены местами его полубайтов) и константы **0Fh**, с сохранением результата этой операции в регистре **W**.
В результате этого, старший полубайт "встает на место" младшего, и в старшем полубайте устанавливается **0**.
По логике операции **И**, результат **побитного И** с нулем, всегда есть ноль, а результат **побитного И** с единицей, повторяет состояние второго "участника" этой логической операции.
Таким образом (**movwf LED7**), в младший полубайт регистра **LED7**, записывается старший полубайт регистра **LED3**, и в старшем полубайте регистра **LED7** устанавливается **0**.
Следующая группа команд работает с младшим полубайтом регистра **LED3**.
В этом случае, менять местами полубайты регистра **LED3** не нужно.
Первая команда (**movfw LED3**) копирует содержимое регистра **LED3** в регистр **W**.
Примечание: **movfw** - не опечатка. Такой команды в распечатке команд Вы не найдете (есть **movwf**), но тем не менее, команда **movfw** работает (можете добавить в список).
Команды **movfw** (название регистра) и **movf** (название регистра)**,W** производят одно и то же действие.
Работа второй команды (**andlw 0Fh**) описана выше.
Работа третьей команды (**movwf LED6**) описана выше, разница только в том, что содержимое регистра **W** копируется не в регистр **LED7**, а в регистр **LED6**.
Таким образом (**movwf LED6**), в младший полубайт регистра **LED6**, записывается младший полубайт регистра **LED3**, и в старшем полубайте регистра **LED6** устанавливается **0**.
Остальные 3 пары групп команд распределения полубайтов работают аналогично.
При распределении полубайтов в регистры **LED0...3**, содержимое их ранее распределенных полубайтов во внимание не берется (они уже распределены), и с ними можно проводить описанные выше операции, не опасаясь "уничтожить нужный" полубайт.
На конечной стадии распределения, младший полубайт регистра **LED0** копируется в него же.
В чем смысл? Ведь содержимое младшего полубайта регистра **LED0**, казалось бы, можно вообще "не трогать" (ничего с ним не делать).
Но в этом случае, старший полубайт регистра **LED0** не будет установлен в ноль.
Поэтому, последняя группа команд распределения должна быть исполнена в полном объеме.
После ее исполнения, рабочая точка программы переходит на команду возврата **return** (ПП **Bin2_10** отработана), и далее, осуществляется возврат на следующую, после команды **call Bin2_10**, команду ПП **START**.
После отработки команд ПП **START**, начинается исполнение ПП динамической индикации (данные для ее работы подготовлены), и так далее.
До следующего "влёта" рабочей точки программы в ПП **Bin2_10**.
"На фоне" сказанного, ПП **adjBCD** является некой "темной лошадкой".
В том смысле, что формально (без "привязки" ко всему остальному), ее работа понятна, но в комплексе со всем остальным, получается довольно-таки объемная и трудно воспринимаемая "круговерть" чисел.
В это можно "въехать", но на данной стадии обучения, какого-то большого, практического смысла в этом нет.
И выбора у меня тоже нет, так как в дальнейшем, без подпрограммы преобразования двоичных чисел в двоично-десятичные, никак не обойтись.
Эта подпрограмма работает четко, и в ее "эксплуатации", никаких сложностей не будет.
Нужно только научиться "трансформировать" ее текст под то, что Вам нужно (а вот в этом есть "могучий", практический смысл).
Как это делается?

Пример

"Переоборудуем" ПП **Bin2_10** под линейку, состоящую из 4-х знакомест.
В этом случае, необходимо произвести следующие действия:

В "шапке" программы

1. Убрать регистры LED4...7, а регистры LED0...3 оставить.
2. Убрать регистры TimerH и TimerHH, а регистры TimerL и TimerM оставить. Таким образом, группа ПП формирования двоичного числа должна быть рассчитана на формирование 2-байтного, двоичного числа, которое должно "лежать" в регистрах TimerL и TimerM.

В рабочей части программы

3. Заменить константу 8x4=.32 на константу 8x2=.16.
4. Убрать 4 команды clrf LED4...7.
5. Убрать 4 команды rlf TimerH,F, rlf TimerHH,F, rlf LED2,F, rlf LED3,F.
6. В группе команд распределения полубайтов, убрать первые 4 (сверху) группы команд распределения.
7. В ПП adjDEC, убрать 3-ю и 4-ю (сверху) группы команд (movlw, movwf, call).

Таким же образом, можно "переоборудовать" ПП **Bin2_10** под любое количество знакомест. Если учесть, что такого же рода "переоборудование" можно произвести и в ПП динамической индикации (об этом говорилось ранее), то в комплексе, получается именно то, что нужно.
А теперь сведем то, с чем мы разбирались по частям, к единому целому.

Получается некая **"универсальная заготовка" с названием Din_Bin.asm** (находится в папке ***"Тексты программ"***).

Она выглядит так:

```
;*******************************************************************************
; Din_Bin.asm   Универсальная "заготовка" программы, включающей в себя группу
;               подпрограмм динамической индикации в комплексе с группой
;               подпрограмм преобразований двоичных чисел в двоично-десятичные
;               для случая 8-разрядной динамической индикации в линейке из 8-ми
;               7-сегментных индикаторов с применением внешнего дешифратора
;               адресного кода 555ИД7.
;*******************************************************************************
;                                 "ШАПКА ПРОГРАММЫ"
;===============================================================================
            LIST        p=16F84a    ; Определение типа микроконтроллера.
            __CONFIG    .....       ; Биты конфигурации.
;.....................................................
;===============================================================================
; Определение положения регистров специального назначения.
;===============================================================================
Indf        equ         00h         ; Регистр Indf.
PC          equ         02h         ; Счетчик команд.
Status      equ         03h         ; Регистр Status.
FSR         equ         04h         ; Регистр FSR.
PortA       equ         05h         ; Регистр управления защелками порта A.
PortB       equ         06h         ; Регистр управления защелками порта B.
;.....................................................
;===============================================================================
; Определение названия и положения регистров общего назначения.
;===============================================================================
LED0        equ         10h         ; Регистр хранения результатов преобразований
                                    ; 1-го двоично-десятичного разряда.
LED1        equ         11h         ; ------- 2-го ------------------------
LED2        equ         12h         ; ------- 3-го ------------------------
LED3        equ         13h         ; ------- 4-го ------------------------
LED4        equ         14h         ; ------- 5-го ------------------------
LED5        equ         15h         ; ------- 6-го ------------------------
LED6        equ         16h         ; ------- 7-го ------------------------
LED7        equ         17h         ; ------- 8-го ------------------------
```

```
Index       equ         0Ch         ; Регистр счетчика количества
                                    ; малых колец индикации.
Count       equ         0Dh         ; Регистр счетчика количества больших колец
                                    ; индикации. Он же - счетчик проходов.
Temp        equ         0Fh         ; Регистр временного хранения данных.
Mem         equ         1Fh         ; Регистр оперативной памяти.
TimerL      equ         1Bh         ; Регистр младшего разряда 4-байтного
                                    ; двоичного числа.
TimerM      equ         1Ch         ; Регистр среднего разряда 4-байтного
                                    ; двоичного числа.
TimerH      equ         1Dh         ; Регистр старшего разряда 4-байтного
                                    ; двоичного числа.
TimerHH     equ         1Eh         ; Регистр самого старшего разряда 4-байтного
                                    ; двоичного числа.
;.....................................................
;=======================================================================
; Определение места размещения результатов операций.
;=======================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;=======================================================================
; Присваивание битам названий.
;=======================================================================
C           equ         0           ; Флаг переноса-заёма.
Z           equ         2           ; Флаг нулевого результата.
;.....................................................
;=======================================================================
; Присвоение константам названий.
;=======================================================================
Const1      equ         Y1          ; Y1 - значение времязадающей константы
                                    ; "грубо" (до .255). Задается программистом.
Const2      equ         Y2          ; Y2 - значение времязадающей константы
                                    ; "точно" (до .255). Задается программистом.
;.....................................................
;=======================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;***********************************************************************

;***********************************************************************
;                           РАБОЧАЯ ЧАСТЬ ПРОГРАММЫ
;***********************************************************************
START       .................................
;           .................................
NEW         call        Bin2_10     ; Условный переход в ПП Bin2_10
                                    ; Адрес следующей команды закладывается в стек
;           .................................
;           .................................
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; ГРУППА ПОДПРОГРАММ 8-РАЗРЯДНОЙ ДИНАМИЧЕСКОЙ ИНДИКАЦИИ без ПП TABLE
; (группа команд ПП TABLE находится в самом конце текста программы).
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; На данный момент, регистры LED0 ... LED7 заполнены двоично-десятичными числами,
; которые необходимо вывести на индикацию (отобразить) в линейку из 8-ми
; 7-сегментных индикаторов.
; На момент начала группы подпрограмм динамической индикации, все прерывания
; должны быть запрещены, все выводы порта B и первые 3 вывода порта A должны быть
; настроены на работу "на выход", работа должна происходить в нулевом банке.
;***********************************************************************
; Подготовка счетчика количества малых колец индикации Index к началу полного
; цикла динамической индикации.
;-----------------------------------------------------------------------
            clrf        Index       ; Сброс в 0 содержимого счетчика малых колец
                                    ; индикации Index.
;-----------------------------------------------------------------------
; Предварительная закладка количества больших колец индикации, которое нужно
```

```
; "пройти" за один полный цикл динамической индикации в регистр Count.
;-------------------------------------------------------------------------------
            movlw       X           ; Запись константы X (количество больших
                                    ; колец индикации, задается программистом),
                                    ; в регистр W.
            movwf       Count       ; Копирование содержимого регистра W,
                                    ; в регистр счетчика количества больших колец
                                    ; индикации Count.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Использование косвенной адресации при работе с таблицей данных.
;===============================================================================
; Подготовка к косвенной адресации: запись в регистр W адреса регистра младшего
; разряда линейки 7-сегментных индикаторов ("привязка" к 7-сегментному
; индикатору, с активации которого начинается полный цикл первого большого
; кольца индикации).
;-------------------------------------------------------------------------------
CYCLE       movlw       LED0        ; Запись в регистр W адреса регистра LED0.
            addwf       Index,W     ; Увеличение адреса регистра LED0 на величину
                                    ; числа, записанного в регистре счетчика
                                    ; количества малых колец индикации Index,
                                    ; С сохранением результата в регистре W.
;-------------------------------------------------------------------------------
; Косвенная адресация.
;-------------------------------------------------------------------------------
            movwf       FSR         ; Копирование содержимого регистра W
                                    ; в регистр FSR.
            movf        Indf,W      ; Копирование содержимого регистра с адресом,
                                    ; записанным в регистре FSR, в регистр W.
            call        TABLE       ; Условный переход (адрес следующей команды
                                    ; закладывается в стек) в ПП TABLE.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Группа команд установки запятой.
;-------------------------------------------------------------------------------
;---> Возврат по стеку из ПП TABLE
            movwf       Temp        ; Копирование содержимого регистра W (7-
                                    ; сегментные коды индицируемых двоично-
                                    ; десятичных чисел) в регистр Temp.
            movlw       5           ; Запись в регистр W константы .05.
            bsf         Status,Z    ; Поднятие флага нулевого результата Z.
            subwf       Index,W     ; Вычесть содержимое регистра W (число .05)
                                    ; из содержимого регистра Index
                                    ; (числа от .00 до .07).
            btfss       Status,Z    ; Проверка состояния флага Z.
            goto        No_Dot      ; Если флаг Z опущен (результат операции
                                    ; не=0), то переход в ПП No_Dot
                                    ; (запятая не выставляется).
            bsf         Temp,7      ; Если флаг Z поднят (результат операции=0),
                                    ; то установка в единицу 7-го бита
                                    ; (установка запятой) регистра Temp.
;-------------------------------------------------------------------------------
; Группа команд вывода десятичной цифры на индикацию.
;-------------------------------------------------------------------------------
No_Dot      movf        Temp,W      ; Копирование содержимого регистра Temp
                                    ; (7-сегментные коды индицируемых двоично-
                                    ; десятичных чисел) в регистр W.
            movwf       PortB       ; Копирование содержимого регистра W
                                    ; в 8 защелок порта B.
;-------------------------------------------------------------------------------
; Группа команд формирования адресного кода управления дешифратором.
;-------------------------------------------------------------------------------
            movf        Index,W     ; Копирование содержимого регистра Index
                                    ; в регистр W.
            movwf       PortA       ; Копирование содержимого регистра W в первые
                                    ; 3 защелки порта A (работа "на выход"),
                                    ; управляющие адресными входами внешнего
                                    ; дешифратора.
```

```
;-------------------------------------------------------------------------
; Группа команд задержки, определяющей время нахождения одного 7-сегментного
; индикатора в активном состоянии (определяющей время прохождения малого кольца
; индикации). "Грубое" формирование времени полного цикла динамической индикации.
;-------------------------------------------------------------------------
            movlw       Const1      ; Запись в регистр W константы Y1 (см"шапку")
            movwf       Temp        ; Копирование содержимого регистра W
                                    ; в регистр Temp.
PAUSE       decfsz      Temp,F      ; Декремент содержимого регистра Temp с
                                    ; сохранением результата в нем же.
            goto        PAUSE       ; Если результат декремента не=0,
                                    ; то переход в ПП PAUSE.
                                    ; Если результат декремента =0,
                                    ; то программа исполняется далее.
;-------------------------------------------------------------------------
; Увеличение на 1 содержимого счетчика количества малых колец индикации Index с
; последующей проверкой результата инкремента на равенство (или нет) числу .08.
;-------------------------------------------------------------------------
            incf        Index,F     ; Увеличение на 1 содержимого регистра Index
                                    ; с сохранением результата в нем же.
            movlw       .08         ; Запись в регистр W константы .08.
            bcf         Status,Z    ; Сброс флага нулевого результата Z.
            subwf       Index,W     ; Вычесть из содержимого регистра Index
                                    ; содержимое регистра W, с сохранением
                                    ; результата в регистре W.
            btfss       Status,Z    ; Результат операции вычитания равен
                                    ; или нет нулю?
            goto        CYCLE       ; Если не =0 (в регистре Index - число не
                                    ; равное 8), то переход к циклу активации
                                    ; следующего по старшинству 7-сегментного
                                    ; индикатора (переход на новое малое кольцо
                                    ; индикации, то есть, в ПП CYCLE).
                                    ; Если =0 (в регистре Index - число равное
                                    ; 8), то программа исполняется далее.
;-------------------------------------------------------------------------
; Начало перехода на новое большое кольцо индикации после того, как
; последовательно активизируются все 8 7-сегментных индикатора линейки
; (после прохождения 8-ми малых колец индикации).
;-------------------------------------------------------------------------
            nop                     ; Выравнивающий NOP.
            clrf        Index       ; Сброс в 0 содержимого регистра Index.
;-------------------------------------------------------------------------
; Уменьшение на 1 содержимого счетчика количества больших колец индикации Count.
;-------------------------------------------------------------------------
            decfsz      Count,F     ; Декремент содержимого счетчика количества
                                    ; больших колец индикации Count, с
                                    ; сохранением результата в нем же.
            goto        CYCLE       ; Если результат декремента не=0, то переход
                                    ; в ПП CYCLE
                                    ; (переход на новое большое кольцо индикации)
                                    ; Если результат декремента =0, то программа
                                    ; исполняется далее (переход на новый полный
                                    ; цикл динамической индикации).
            nop                     ; Выравнивающий NOP.
;=========================================================================
; Группы подпрограмм и команд, осуществляющие различные операции.
;=========================================================================
; "Точное" формирование времени полного цикла динамической индикации (если
; требуется точно калиброванное время полного цикла динамической индикации для
; использования его в качестве измерительного интервала).
;-------------------------------------------------------------------------
            movlw       Const2      ; Запись в регистр W константы Y2 (см"шапку")
            movwf       Temp        ; Копирование содержимого регистра W
                                    ; в регистр Temp.
PAUSE_1     decfsz      Temp,F      ; Декремент содержимого регистра Temp с
                                    ; сохранением результата в нем же.
```

```
            goto        PAUSE_1     ; Если результат декремента не=0,
                                    ; то переход в ПП PAUSE_1.
                                    ; Если результат декремента =0,
                                    ; то программа исполняется далее.
;-------------------------------------------------------------------------------
; Гашение активного разряда линейки.
;-------------------------------------------------------------------------------
            movlw       0           ; Запись в регистр W константы .00.
            movwf       PortB       ; Сброс в 0 всех защелок порта B.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; ГРУППА ПОДПРОГРАММ ФОРМИРОВАНИЯ 4-БАЙТНОГО ДВОИЧНОГО ЧИСЛА.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;....        ....................................
;            ....................................    ЗДЕСЬ МОГУТ БЫТЬ РЕАЛИЗОВАНЫ
;            ....................................       РАЗЛИЧНЫЕ "ФАНТАЗИИ"
;            ....................................          ПРОГРАММИСТА

; К этому моменту 4-байтное двоичное число (в регистре
; TimerL/TimerM/TimerH/TimerHH) должно быть сформировано для последующей
; обработки его в группе подпрограмм преобразования 4-байтных двоичных чисел
; в 8-разрядные десятичные числа.
            goto        NEW         ; Безусловный переход на метку NEW
                                    ; подпрограммы START, то есть, начало нового
                                    ; полного цикла "программы".
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; ГРУППА ПОДПРОГРАММ ПРЕОБРАЗОВАНИЯ 4-БАЙТНЫХ ДВОИЧНЫХ ЧИСЕЛ
; В 8-РАЗРЯДНЫЕ ДЕСЯТИЧНЫЕ ЧИСЛА.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Подготовка к преобразованию.
;===============================================================================
Bin2_10     bcf         Status,C    ; Сброс флага переноса-заёма.
            movlw       .32         ; Запись в регистр Count числа проходов
            movwf       Count       ; преобразования, равного суммарному
                                    ; количеству битов многоразрядного регистра
                                    ; TimerL/TimerM/TimerH/TimerHH (8*4=32).
            clrf        LED0        ; Сброс в 0 содержимого регистра LED0.
            clrf        LED1        ; -------"------ LED1.
            clrf        LED2        ; -------"------ LED2.
            clrf        LED3        ; -------"------ LED3.
            clrf        LED4        ; -------"------ LED4.
            clrf        LED5        ; -------"------ LED5.
            clrf        LED6        ; -------"------ LED6.
            clrf        LED7        ; -------"------ LED7.
;-------------------------------------------------------------------------------
; Примечание: процесс преобразования заканчивается при уменьшении числа проходов
;             преобразования, которые заложены в регистр Count (.32), до нуля.
;===============================================================================
; Циклический сдвиг влево.
;===============================================================================
Loop16      rlf         TimerL,F    ; Циклический сдвиг влево 4-байтного двоичного
            rlf         TimerM,F    ; числа, записанного в группе регистров
            rlf         TimerH,F    ; TimerL/TimerM/TimerH/TimerHH,на одну позицию
            rlf         TimerHH,F   ; через бит C регистра STATUS.

            rlf         LED0,F      ; Циклический сдвиг влево 4-байтного двоичного
            rlf         LED1,F      ; числа, записанного в группе регистров
            rlf         LED2,F      ; LED0/LED1/LED2/LED3, на одну позицию через
            rlf         LED3,F      ; бит C регистра STATUS.

            decfsz      Count,F     ; Декремент (-1) содержимого регистра Count с
                                    ; сохранением результата в нем же.
            goto        adjDEC      ; Если результат не=0, то переход в ПП adjDEC
                                    ; Если  результат =0, то программа исполняется
                                    ; далее.
;===============================================================================
; Поразрядное распределение содержимого регистров LED0...3 (обоих
```

```
; полубайтов) по младшим полубайтам регистров LED0...7.
;===============================================================================
            swapf       LED3,W      ; Запись старшего полубайта LED3
            andlw       0Fh         ; в младший полубайт LED7.
            movwf       LED7        ; -------------------------------

            movfw       LED3        ; Запись младшего полубайта LED3
            andlw       0Fh         ; в младший полубайт LED6.
            movwf       LED6        ; -------------------------------

            swapf       LED2,W      ; Запись старшего полубайта LED2
            andlw       0Fh         ; в младший полубайт LED5.
            movwf       LED5        ; -------------------------------

            movfw       LED2        ; Запись младшего полубайта LED2
            andlw       0Fh         ; в младший полубайт LED4.
            movwf       LED4        ; -------------------------------

            swapf       LED1,W      ; Запись старшего полубайта LED1
            andlw       0Fh         ; в младший полубайт LED3.
            movwf       LED3        ; -------------------------------

            movfw       LED1        ; Запись младшего полубайта LED1
            andlw       0Fh         ; в младший полубайт LED2.
            movwf       LED2        ; -------------------------------

            swapf       LED0,W      ; Запись старшего полубайта LED0
            andlw       0Fh         ; в младший полубайт LED1.
            movwf       LED1        ; -------------------------------

            movfw       LED0        ; Запись младшего полубайта LED0
            andlw       0Fh         ; в младший полубайт LED0.
            movwf       LED0        ; -------------------------------
;-------------------------------------------------------------------------
; Конец распределения. В младших полубайтах регистров LED0...7 установлены
; двоично-десятичные числа в порядке возрастания разрядности.
; Старшие полубайты = 0.
;-------------------------------------------------------------------------
            return                  ; Переход по стеку в группу подпрограмм
                                    ; 8-разрядной динамической индикации.
;===============================================================================
; Запись в регистр FSR адресов регистров LED0...3 для дальнейшей косвенной
; адресации к ним в ПП adjBCD.
; Переход к обработке следующего LED - после возврата по стеку.
;===============================================================================
adjDEC      movlw       LED0        ; Запись в регистр FSR, через регистр W,
            movwf       FSR         ; адреса регистра LED0 с дальнейшим переходом
            call        adjBCD      ; в ПП adjBCD (адрес следующей команды
                                    ; закладывается в стек).
;---> Возврат по стеку из ПП adjBCD.
            movlw       LED1        ; ----------------------------
            movwf       FSR         ; То же самое для регистра LED1.
            call        adjBCD      ; ----------------------------
;---> Возврат по стеку из ПП adjBCD.
            movlw       LED2        ; ----------------------------
            movwf       FSR         ; То же самое для регистра LED2.
            call        adjBCD      ; ----------------------------
;---> Возврат по стеку из ПП adjBCD.
            movlw       LED3        ; ----------------------------
            movwf       FSR         ; То же самое для регистра LED3.
            call        adjBCD      ; ----------------------------
;---> Возврат по стеку из ПП adjBCD.
            goto        Loop16      ; Проход всех LED (с LED0 по LED3). Переход в
                                    ; ПП Loop16, то есть на следующее кольцо
                                    ; числовых преобразований.
;===============================================================================
```

```
; Основные операции преобразования двоичных чисел в двоично-десятичные:
; операции сложения LED0...3 и констант 03h,30h с условиями по 3-му и 7-му битам.
;=============================================================================
adjBCD      movlw     3           ; Сложить содержимое текущего LED (LED0...3) с
            addwf     0,W         ; числом 03h, с записью результата операции,
            movwf     Mem         ; через регистр W, в регистр Mem.

            btfsc     Mem,3       ; Анализ состояния 3-го бита регистра Mem.
            movwf     0           ; Если бит № 3 =1, то содержимое регистра Mem
                                  ; копируется в текущий LED.
            movlw     30          ; Если бит №3 =0, то содержимое текущего LED
            addwf     0,W         ; складывается с константой 30h, с последующей
            movwf     Mem         ; записью результата операции,через регистр W,
                                  ; в регистр Mem.
            btfsc     Mem,7       ; Анализ состояния 7-го бита регистра Mem.
            movwf     0           ; Если бит №7 =1, то содержимое регистра Mem
                                  ; копируется в текущий LED.
            retlw     0           ; Если бит №7 =0, то регистр W очищается и
                                  ; происходит возврат по стеку в ПП adjDEC.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; ГРУППА КОМАНД ПРЕОБРАЗОВАНИЯ ДВОИЧНО-ДЕСЯТИЧНОГО КОДА В КОД 7-СЕГМЕНТНОГО
;     ИНДИКАТОРА (относится к группе подпрограмм динамической индикации).
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
TABLE       addwf     PC,F        ; Содержимое счетчика команд PC увеличивается
                                  ; на величину содержимого аккумулятора W.
            retlw     b'00111111' ; ..FEDCBA = 0   Происходит скачек по таблице
            retlw     b'00000110' ; .....CB. = 1   на строку со значением,
            retlw     b'01011011' ; .G.ED.BA = 2   записанным в аккумуляторе,
            retlw     b'01001111' ; .G..DCBA = 3   и далее - возврат по стеку.
            retlw     b'01100110' ; .GF..CB. = 4
            retlw     b'01101101' ; .GF.DC.A = 5
            retlw     b'01111101' ; .GFEDC.A = 6
            retlw     b'00000111' ; .....CBA = 7
            retlw     b'01111111' ; .GFEDCBA = 8
            retlw     b'01101111' ; .GF.DCBA = 9
;*****************************************************************************
            end                   ; Конец программы.
```

Все что Вы видите, было "разобрано" ранее.
Сразу бросается в глаза "разросшаяся шапка" программы, что вполне естественно.
Регистр **Count** так же, как и регистр **Temp**, задействуется многократно (в данном случае, 2 раза).
Если Вы проассемблируете текст программы **Din_Bin.asm**, то получите сообщение о 7-ми ошибках, что вполне закономерно, так как в тексте этой программы, значения трех констант выражены в символьном виде (**X**, **Y1**, **Y2**).
Это 3 "ненормальности", которые **MPLAB** "воспринимает" как ошибки.
Остальные 4 ошибки связаны с "непонятностью" того, к чему обращается директива **CONFIG** (**....**) и "непонятностью" первой команды ПП **START** (**....**).
Если символьные константы **X**, **Y1**, **Y2** будут заменены на числа, директива **CONFIG** будет обращаться к "нормальному" числу (например 03FF1H) и будет определена 1-я команда ПП **START**, то никаких сообщений об ошибках не будет.
Если Вы хотите перейти к другому количеству знакомест линейки, то руководствуйтесь изложенной ранее информацией, в которой "расписан" принцип такого перехода.
Числовые значения констант **X**, **Y1**, **Y2** определяет программист, исходя из стоящих перед ним задач.
На что влияют эти числовые значения, указано ранее.
Если изложенные выше методики "трансформации" этой универсальной "заготовки" освоены и понятны, то после достижения желаемого, ее нужно "доукомплектовать" подпрограммой формирования **N**-байтного (**N** определяет программист), двоичного числа.
"Конструкция" этой ПП может быть различной.
В **N**-байтное, двоичное число можно "загнать" любой параметр внешнего (по отношению к микроконтроллеру) сигнала.
Выбор богатейший.

Таким образом, в том, что я назвал ***Группой подпрограмм формирования 4(N) - байтного двоичного числа***, "сконцентрирован" очень объемный класс задач, на решение которых, в большинстве случаев, и направлены основные усилия программиста.
Объем этого класса задач велик и могуч, но "набор приемов", который позволяет эффективно их решать, конечен.
Освоение этих приемов - вполне посильная работа.
Вы познакомились с их частью (многие приемы универсальны), а это уже вполне приличный старт.
Если Вы, с первого захода, поняли хотя бы 50% из того, о чем шла речь, то это уже несомненный успех.
Программирование – "дело наживное" и совсем не "шапкозакидательское".
Если не получается, то повторите "заходы", и при наличии "упёртости", рано или поздно, "все встанет на свои места".
Противник грозный и очень достойный.
Он, по определению, будет сопротивляться. И это нужно как следует понимать.
Несколько "голов", у этого "многоголового дракона, мы посшибали".
Давайте "посшибаем" еще.

17. Принцип счета. Работа с таймером TMR0. Принцип установки групп команд счета в текст программы.

Как это не странно, но изложенная ниже информация, почему-то, образно выражаясь, является "страшно секретной".
В том смысле, что в свое время, мне так и не удалось найти более-менее "внятной" информации по высокоскоростному счету.
Это вынудило хорошенько "поднапрячься".
А куда деваться?
Заниматься программированием и игнорировать "базу", это нонсенс.
Ниже предпринята попытка достаточно подробного (как мне кажется) описания механизма низкоскоростного и высокоскоростного счета, рассчитанная на начинающих программистов, усвоивших предыдущие разделы.

Низкоскоростной счет

Первичная информация о таймере **TMR0** и принципе счета, основанном на его применении, была изложена ранее.
Казалось бы, все просто: при работе от внешнего источника импульсов, подсчитывается количество переполнений.
Оно умножается на 256, и к этому произведению приплюсовывается число, находящееся в **TMR0** на момент окончания счета.
Примечание: речь о низкоскоростном счете идет в том случае, когда для подсчета импульсов применяется только **TMR0** (без предделителя).
Сигнал от внешнего источника сигнала, на вход **TMR0**, подается через синхронизатор.
Функцией синхронизатора является синхронизация сигнала, поступающего от внешнего источника сигнала, с внутренней, тактовой частотой м/контроллера.
В связи с этим, период сигнала, поступающего от внешнего источника сигнала, не должен быть менее 4-х периодов внутреннего такта.
Например, при применении кварца номиналом 4 Мгц., период сигнала, поступающего от внешнего источника сигнала, не должен быть менее 1 мкс., и это есть "граница", за которую "заходить" нельзя.
Таким образом, при использовании кварца номиналом 4 Мгц., **TMR0** не может работать с частотами выше 1 Мгц.
Плюс, при приближении к этой "границе", нужно будет организовывать достаточно частые проверки на переполнение **TMR0**, и для хранения результатов подсчетов переполнений, одного регистра будет маловато.
Не смотря на то, что по способу своей организации, низкоскоростной счет достаточно прост (задействуется только **TMR0**), но он, по причинам указанным выше, не находит широкого применения.
Каков выход?
Выход → в использовании предделителя.
Например, если использовать предделитель с максимальным коэффициентом деления (256), то переполнение **TMR0** наступит через 256x256=65536 импульсов.
Если измерять частоту 30 Мгц., то 65536 импульсов "проскочат" примерно за 2,2 мс., что уже есть "зер гут" (прикиньте, сколько команд можно исполнить за 2200 мкс.).
По причине того, что в случае использования предделителя, счетным входом ПИКа является не счетный вход **TMR0**, а вход предделителя, условия нормального функционирования **TMR0** обеспечиваются даже при подсчете импульсов с большой частотой следования.
И количество проверок на переполнение **TMR0** резко сокращается.
Мало того, при таком "раскладе", можно даже поднять частотную, "программную границу" (максимальную скорость счета, на которую рассчитана программа) выше частотной, "аппаратной границы" ПИКа (максимальной скорости счета, на которую рассчитан ПИК).

Высокоскоростной счет

Для того чтобы было понятно, с чем мы имеем дело, есть смысл кое-что напомнить и добавить.
Таймер **TMR0** представляет собой делитель на **256**.

Это означает, что "чистый" **TMR0** (без использования предделителя) будет переполняться каждый раз после поступления на его счетный вход (вывод **RA4/TOCKI**) "пачки" из 256 импульсов (при переходе от **.255** к **.00**).
TMR0 считает только в одну сторону (реверса нет), а именно, в сторону увеличения.
Если в качестве "предмета счета", выбран внешний тактовый сигнал, присутствующий на выводе **RA4/TOCKI**, то подсчитывается количество импульсов, формируемых внешним источником импульсов.
Примечание: "внешний тактовый сигнал" буду называть "внешним тактом". Так короче.
В случае выбора внешнего такта, импульсы внутреннего такта (**CLKOUT**) не подсчитываются (и наоборот).
TMR0 всегда включен. Выключить его нельзя.
По этой причине, для того чтобы, при наличии импульсной последовательности на выводе **RA4/TOCKI** (выбран внешний такт), остановить или разрешить счет, нужно заблокировать или разблокировать счетный вход ПИКа (вывод **RA4/TOCKI**).
То есть, примитивно выражаясь, его нужно либо "закоротить" на корпус, либо снять этот "коротыш" (с этим, детально разберемся позднее, "а пока и так сойдет").
Для этого, выводы **RA3** (блокировка/разблокировка) и **RA4/TOCKI** (счетный вход ПИКа) электрически соединяются между собой, и блокировка осуществляется программно.
То есть, если на выводе **RA3** (можно использовать и другой вывод порта) устанавливается **0** (блокировка), то это соответствует остановке счета, а если **1** (разблокировка), то это соответствует разрешению счета.
Критерием переполнения **TMR0** является **флаг прерывания по переполнению TMR0** (бит **№2** регистра **INTCON** с названием **T0IF**).
Поднятие этого флага (**1**) свидетельствует о факте переполнения **TMR0**, а его "опущение" (**0**. Извиняюсь за неблагозвучие) свидетельствует о том, что переполнения **TMR0** нет.
В определении флага **T0IF**, упоминаются прерывания, но этот флаг работает независимо от того, разрешены или запрещены прерывания по переполнению **TMR0**.
То есть, если прерывания по переполнению **TMR0** запрещены, то флаг **T0IF** тоже будет "полноценно" работать.
Флаг **T0IF** относится к флагам 2-й группы.
То есть, после того как он поднялся (**1**), и факт этого события подсчитан ("задокументирован"), в конце процедуры проверки **TMR0** на переполнение, флаг **T0IF** нужно опустить (**0**), а иначе, результат следующей(их) проверки(ок) **TMR0** на переполнение, будет необъективен.
Напоминаю о том, что все флаги 2-й группы опускаются только программно.
Разбираемся дальше.
Для того чтобы разговор был предметным, речь пойдет о частотомере.
"Привяжусь" к 3-байтному результату подсчета импульсов.
Результаты этого подсчета будут записываться в 3-байтный регистр **TimerH/TimerM/TimerL**.
Иными словами, после завершения подсчета, в этом регистре, с соблюдением порядка старшинства, "осядет" 3-байтное, двоичное число результата измерения частоты.
Интервал времени измерения: 0,1 сек. (100000 мкс.).
Максимальная, измеряемая частота: 30 Мгц. (паспортная, максимальная частота, которую "способен воспринять" счетный вход PIC16F84A).
Разрядность линейки 7-сегментных индикаторов: 7 десятичных разрядов (знакомест).
Погрешность измерения: 10 гц.
Рассмотрим случай подсчета импульсов импульсной последовательности с частотой 30 Мгц.
Обращаю Ваше внимание на то, что в этом случае, речь идет не о подсчете 30 000 000 импульсов, а о подсчете **3 000 000** импульсов, так как интервал времени измерения составляет не 1 сек., а **0,1 сек.** (отсюда и погрешность в 10 гц.).
С учетом сказанного выше, для такой высокой частоты, вариант счета с использованием "чистого" **TMR0** не подходит.
Значит, необходимо замедлить счет **TMR0**.
То есть, нужно сделать так, чтобы счетный вход **TMR0** "реагировал" не на каждый импульс, поступающий на вывод **RA4/TOCKI**, а например, на каждый 10-й, 20-й, 34-й, ... и т.д. (цифры "взяты с потолка").
Соответственно, в 10, 20, 34, ... раз увеличится и время одного цикла переполнения, что и требуется (проверки на переполнение не будут очень частыми).
Для этого, между выводом **RA4/TOCKI** (счетным входом ПИКа) и счетным входом **TMR0**,

необходимо включить делитель на 10, 20, 34, ...
Такой делитель в ПИКе имеется.
Он называется **предделителем**.
Посмотрите в распечатку регистра **OPTION**.
Для того чтобы обеспечить требуемый режим работы устройства, до "влёта" в цикл счета, нужно:
- установить бит **№5** в **1** (***выбор внешнего такта с вывода* RA4/TOCKI**),
- бит **№4**, обычно, устанавливают в **0** (***приращение по переднему фронту***),
- бит **№3** устанавливается в **0** (***предделитель включен перед* TMR0**),
- а с битами **№№ 0 ... 2** (***коэффициент деления предделителя***) давайте разбираться.

Какой коэффициент деления предделителя установить?
Выбираем максимально возможный Кделения предделителя (**256**).
Это соответствует заданию наивысшего, частотного "потолка".
Можно "привязаться" и к полубайту (128) и к четверти байта (64), но при этом наживаются никому не нужные проблемы, связанные с необходимостью работы либо с полубайтами, либо с четвертями байта, со всеми вытекающими, не очень-то "комфортными", последствиями.
Итак, в битах **№№ 0 ... 2** выставляем единицы (***Кделения предделителя* = 256**).
В этом случае, переполнение **TMR0** будет происходить не после подсчета "пачки", состоящей из 256-ти импульсов, а после подсчета "пачки", состоящей из
256x256=**65536**-ти импульсов.
Таким образом, за время "прохождения" **3 000 000** импульсов, произойдет **45** переполнений **TMR0** (по принципу "недопущения перебора", округлено в сторону уменьшения).
65536x45=**2949120**.
Остаток: 3000000-2949120=**50880**.
Обратите внимание на то, что число **.45** не выходит за "верхнюю границу" числового диапазона байта (не более **.255**), и оно "уместится" в одном байте.
Таким образом, речь идет о том, что в случае организации высокоскоростного счета, можно работать (программно) с частотами более высокими, чем 30 Мгц.
30 Мгц. это "официальный, паспортный потолок" PIC16F84A.
Если будет использован более быстродействующий ПИК, то программа может обеспечить его работу на частотах более высоких, чем 30 Мгц.
Вернемся к математике.
Имеется остаток равный **50880**.
На 65536 его делить нельзя, так как получится число меньше единицы, значит делим на 256.
С учетом соблюдения все того же принципа "недопущения перебора", получится следующее:
256x**.198**=**50688**.
Остаток: 50880-50688=**.192**.
Еще раз обратите внимание на то, что и число **.198**, и число **.192** не выходят за пределы числового диапазона байта, и каждое из них можно отобразить в одном байте.
Итожим: **3 000 000=256x256x45+256x198+192**
Вы видите математическую форму разложения десятичного числа **3 000 000**.
Для того чтобы убедиться в том, что между этой формой представления числа (используемые числа - десятичные) и двоичной (бинарной) формой представления этого же числа, имеется однозначное соответствие, в программке конвертора систем исчислений, имеющейся у Вас, "настучите" число **3000000**.
Вы увидите, что десятичное число **3000000**, в бинарном виде, выглядит так:
00101101 11000110 11000000
А теперь по частям:
- старший байт: 256x256x45=**2949120** (**00101101 00000000 00000000**), (**00101101=.45**),
- средний байт: 256x198=**50688** (**00000000 11000110 00000000**), (**11000110=.198**),
- младший байт: 192 (**00000000 00000000 11000000**), (**11000000=.192**).

Остается только "закрепить"
- **за старшим байтом** (число **.45**) → регистр **TimerH**,
- **за средним байтом** (число **.198**) → регистр **TimerM**,
- **за младшим байтом** (число **.192**) → регистр **TimerL**.

Каждое из этих трех чисел отображается в одном байте.
3-байтное, двоичное число, записанное в регистре **TimerH/TimerM/TimerL**, будет

соответствовать 7-разрядному, десятичному числу **3 000 000**.
Переполнение **TMR0** будет происходить 3000000/65536=**45,776367 раз в 0,1 сек.** (или 457,76367 раз за 1 сек.), что соответствует числовому значению интервала времени между двумя соседними переполнениями, **2184,5 мкс.**
В этом интервале времени можно исполнить сотни команд.
И это с учетом того, что речь идет об измерении достаточно высокой частоты (30 Мгц.).
Этот интервал времени "прекрасно вписывается" **в малое кольцо динамической индикации** (процессы счета и динамической индикации совмещены).
В данном случае, время отработки одного цикла малого кольца динамической индикации не должно превышать 2184,5x2=**4369 мкс.**, так как в противном случае, возникнет погрешность подсчета (2 переполнения **TMR0** будут считаться за одно).
Вывод: в данном случае, проверки на переполнение **TMR0** нужно производить чаще, чем 1 раз за **4369 мкс.**
Еще один вывод: для того чтобы отодвинуть верхнюю, программную границу (она определяет быстродействие программы, а не ПИКа) полосы частот, в пределах которой производится счет, вверх по оси частот, необходимо уменьшить время полного цикла малого кольца динамической индикации.
В этом случае, нужно не забывать о том, что это влияет и на время отработки цикла динамической индикации, и на величину интервала времени измерения.
После подобного рода "катаклизма" (если он имеет место быть), нужно производить коррекцию временных характеристик программы.
Итак, команды 1-й проверки **TMR0** на переполнение "врезаются" в малое кольцо динамической индикации, а результат этих подсчетов "оседает" в регистре **TimerH**.
Вопрос: "Каким образом происходит подсчет"?
Ответ: каждый раз, при наличии факта переполнения **TMR0** (флаг **T0IF** поднят), содержимое регистра **TimerH** инкрементируется.
Количество таких инкрементов напрямую зависит от частоты импульсной последовательности, подаваемой на вывод **RA4/TOCKI** (на вход предделителя).
В зависимости от нее (от частоты), переполнения **TMR0** будут происходить не на каждом "витке" малого кольца динамической индикации (такое возможно только при приближении к верхней границе быстродействия программы), а например, 1 раз за 5, 65, 231, 564 и т.д. "витка" (цифры "взяты с потолка").
Если речь идет о 1-й проверке **TMR0** на переполнение, то должна быть и вторая?
Так оно и есть.
Дело в том, что между группой команд 1-й проверки **TMR0** на переполнение и командой окончания счета, располагается группа команд точной "доводки" интервала времени измерения и группа команд завершения измерения.
В интервале времени их отработки, может произойти переполнение **TMR0**, которое не будет подсчитано.
Следовательно, сразу же после окончания счета, необходимо произвести еще одну проверку на переполнение **TMR0** (№2).
Если, за указанный выше промежуток времени, произошло переполнение **TMR0** (из-за малой его величины, количество переполнений не может быть больше одного), то содержимое регистра **TimerH** инкрементируется, а если не произошло, то содержимое регистра **TimerH** не меняется.
Итак, с регистром старшего разряда **TimerH**, в основном, разобрались.
Для удобства, будем считать, что старший байт числа **01101101 00000000 00000000** (2949120) записан в регистр **TimerH**, и он "лежит в оперативных закромах" ("ждет своего часа").
Итог по регистру **TimerH**.
В ходе исполнения процедур последовательных проверок переполнения **TMR0** (проверка №1), числовое значение байта регистра **TimerH** будет увеличиваться.
Интенсивность этого увеличения зависит от значения измеряемой частоты.
После разрешения счета, на каждом "витке" малого кольца динамической индикации, происходит проверка переполнения **TMR0** №1, за счет которой и осуществляется основной "прирост" числового значения байта регистра **TimerH**.
После того как счет запрещается, осуществляется 2-я проверка переполнения **TMR0**, которая может либо привести, либо не привести к единственному инкременту содержимого регистра **TimerH**.

После проверки переполнения **TMR0** №2, в регистре **TimerH**, будет окончательно сформировано числовое значение старшего байта 3-байтного двоичного числа результата подсчета (измерения).
Ранее, для определенности, мы условились, что это число **0010 1101** (.**45**).
Теперь обратим внимание на регистр среднего разряда **TimerM**.
Итак, счет закончен, и в регистр **TimerH** записано количество переполнений **TMR0**, произошедших в интервале времени измерения.
Вопрос: что нужно записать в регистр среднего разряда **TimerM**?
Ответ: в него нужно записать содержимое **TMR0** на момент окончания счета.
Эта операция простая: содержимое **TMR0**, через регистр **W**, просто копируется в регистр **TimerM**.
Таким образом, в рассматриваемом случае, в средний байт 3-байтного двоичного числа (**TimerM**) запишется число **1100 0110** (**.198**).
В "связке" с регистром **TimerH**, получилось это:

01101101 1100 0110 00000000

Переходим к регистру младшего разряда **TimerL.**
Вопрос "на засыпку": если количество переполнений **TMR0** записалось в регистр **TimerH**, а содержимое **TMR0** (на момент окончания счета) скопировалось в регистр **TimerM**, то что же тогда записывать в регистр **TimerL**?
Ответ: если не найти то "место", в котором "лежит хвостик" разложения значения подсчитываемого количества импульсов (в нашем случае, число **.192**), то записывать в регистр **TimerL** нечего.
И в самом деле, **TMR0** "выжат как мочалка" и "выжимать" из него нечего, но из предделителя кое-что "выжать" можно.
Именно в байте предделителя и "лежит этот хвостик".
А вот здесь-то и начинается самое интересное.
Речь идет о такой "головоломке", как подпрограмма досчета.
Из ее названия понятно, что организуется дополнительная процедура досчета.
Именно результат работы этой ПП и будет записываться в регистр **TimerL**.
Давайте разберемся что это такое.
Сначала в принципе, а затем и более детально.

Принцип работы подпрограммы досчета

Досчет это счет с предустановкой, в пределах одного цикла счета.
Перед исполнением ПП досчета, содержимое регистра **TimerL** необходимо сбросить в ноль (подготовка к досчету).
Так как то, что нужно подсчитать, фиксировано, то необходимо организовать "принудительный" счет.
Для этого нужно сформировать серию коротких, счетных импульсов.
Импульс формируется путем кратковременной установки единичного уровня на выводе блокировки **RA3**, который электрически соединен с выводом счетного входа **RA4/TOCKI**.
То есть, счетный импульс формируется программными средствами.
В нашем случае (приращение **TMR0** по переднему фронту), в момент начала формирования счетного импульса, формируется активный перепад (переход от **0** к **1**), поступающий на счетный вход ПИКа (то есть, на вход предделителя), который и будет подсчитан.
Возникает вопрос: "А не будут ли, во время формирования единичных уровней счетных импульсов, паразитно подсчитываться импульсы от внешнего источника импульсов, ведь это плохо отразится на итоговом результате подсчета"?
Ответ: импульсы, от внешнего источника импульсов, подсчитываться не будут, так как в интервале времени исполнения ПП досчета, блокировочный вывод **RA3** настроен на работу "на выход", что соответствует подключению, к счетному входу ПИКа, низкоомного выхода третьей защелки порта А.
В этом случае, на счетном входе ПИКа, амплитуда импульсов, поступающих (через гасящий резистор) от внешнего источника импульсов, за счет шунтирования счетного входа ПИКа низким выходным сопротивлением третьей защелки порта А, будет резко снижена, и она (амплитуда) окажется существенно ниже порога "срабатывания" счетного входа ПИКа.

По этой причине, ***счетный вход ПИКа перестанет "реагировать" на импульсы от внешнего источника, но будет "реагировать" на счетные импульсы.***
Примечание: в процессе основного счета, который происходил до "влёта" в процедуру досчета, такое шунтирующее влияние отсутствует, так как в интервале времени основного счета, вывод **RA3** настроен на работу "на вход" (выход защелки отключен от вывода **RA3**).
После формирования счетного импульса, может наступить 2 события:
- содержимое **TMR0** не инкрементировалось,
- содержимое **TMR0** инкрементировалось.

Пока, просто примите это к сведению.
До сих пор, я не обращал Вашего внимания на байт, "дислоцирующийся" в предделителе. Пришла пора с ним разобраться.
Байт предделителя, по отношению к байту **TMR0**, является **младшим**, и этим все сказано.
И в самом деле, где, как не в байте предделителя, находиться "хвостику" результата измерения? Другого не дано.
То есть, ***для получения десятичного "эквивалента" 3-байтного двоичного числа, точно равного истинному результату измерения (подсчета), содержимое байта предделителя нужно записать в регистр*** **TimerL**.
Это и есть та задача, которую нужно решить.
Таким образом, в байте предделителя "лежит" число **.192** (условились ранее), и нужно каким-то образом "переправить" это число, из байта предделителя, в байт регистра **TimerL**.
Задача, казалось бы, простая, но это не так.
Так как предделитель не является регистром, отображаемым в области оперативной памяти, то при помощи стандартной процедуры записи (через регистр **W**), "переправить" его содержимое, в регистр **TimerL**, нельзя.
Самое разумное, что можно сделать, так это инкрементировать содержимое байта предделителя до тех пор, пока число **.255** сменится на число **.00**.
В этом случае, произойдет инкремент содержимого регистра **TMR0**, что и будет являться критерием конца досчета.
Факт инкремента содержимого регистра **TMR0** укажет на то, что байт предделителя инкрементирован количество раз, равное разнице между числом 256 и числом, находящимся в байте предделителя на момент окончания счета (то есть, числом, которое нужно "переправить" в регистр **TimerL**).
Примечание: приращение содержимого **TMR0** не будет влиять на результат основного счета, так как оно происходит после того, как этот результат сохранен в регистрах **TimerH** и **TimerM**.
Подсчитав количество инкрементов байта предделителя (от первого инкремента и до обнаружения факта приращения содержимого регистра **TMR0**) и осуществив числовое преобразование, можно точно восстановить числовое значение байта предделителя на момент окончания основного счета ("хвостик").
Обращаю Ваше внимание на то, что в данном случае, речь идет о досчете, а не о счете.
Счет производится в интервале времени измерения (основной счет), на момент окончания которого, в байте предделителя фиксируется некое число.
Досчет производится вне интервала времени измерения и "принудительно" (программная "симуляция" счета).
Если добавить результат досчета к результату основного счета, то программная погрешность результата измерения не будет превышать 10-ти герц.
"Привяжемся" к рассматриваемому случаю (подсчет **3 000 000** импульсов за **0,1 сек**).
Итак, в байте предделителя "лежит" число **.192**.
Задача: нужно "переправить" число **.192**, из байта предделителя, в регистр **TimerL**.
Математически это выглядит так:
1. 256-**192**=**.64** (значение результата досчета).
2. Переводим десятичное число **.64** в бинарную форму: **01000000**.
3. Инвертируем все биты числа **01000000**.
Получаем: **10111111**, что соответствует числу **.191**.
4. 191+1(инкремент)=**.192** → **TimerL**.
Итог: число **.192** "переправлено", из байта предделителя, в регистра **TimerL**.
Для того чтобы Вы убедились, что и в других случаях, происходит то же самое, приведу еще один пример.
Например, считается **1 500 000** импульсов за **0,1 сек.**
Делаем разложение: **256x256x22+256x227+96**

1. 256-**96**=**.160** (значение результата досчета).
2. Переводим десятичное число **.160** в бинарную форму: **10100000**.
3. Инвертируем все биты числа **10100000**.
   Получаем: **01011111**, что соответствует числу **.95**.
4. 95+1(инкремент)=**.96**.

Пока хватит ("артподготовка"). С деталями разберемся позднее.

Краткий, общий итог

Наибольший практический интерес представляет принцип организации высокоскоростного счета.
При этом, перед TMR0, включается предделитель, на вход которого (предделителя)***, с выхода внешнего источника импульсов, подается последовательность импульсов, количество которых нужно посчитать в интервале времени счета. Организуется 3-байтный*** (в рассматриваемом случае) ***регистр, в старший разряд которого записывается количество переполнений TMR0, произошедших за интервал времени счета, в средний разряд записывается число, зафиксировавшееся в TMR0 на момент окончания счета, в младший разряд записывается число, зафиксировавшееся в байте предделителя на момент окончания счета.***

Для детального разбирательства с механизмом высокоскоростного счета, используется файл **Tmr0.asm** (находится в папке ***"Тексты программ"***).

Это выглядит так:

```
;*******************************************************************************
; Tmr0.asm   ИЛЛЮСТРАЦИЯ ПРИНЦИПА ВЫСОКОСКОРОСТНОГО СЧЕТА ИМПУЛЬСОВ
;      ОТ ВНЕШНЕГО ИСТОЧНИКА ИМПУЛЬСОВ С ИСПОЛЬЗОВАНИЕМ ТАЙМЕРА TMR0
;*******************************************************************************
; "Заготовка" "программы", в которой осуществляется высокоскоростной счет
; импульсов от внешнего источника импульсов с использованием таймера TMR0.
; Используется PIC16F84A.
; Корабельников Е.А. г.Липецк  http://ikarab.narod.ru  E-mail: karabea@lipetsk.ru
;*******************************************************************************
;                                  "ШАПКА ПРОГРАММЫ"
;*******************************************************************************
;.............................................................
;.............................................................
;==============================================================================
; Определение положения регистров специального назначения.
;==============================================================================
Tmr0        equ         01h         ; Регистр TMR0.
OptionR     equ         01h         ; Регистр Option - банк 1.
Status      equ         03h         ; Регистр Status.
PortA       equ         05h         ; Регистр PortA.
TrisA       equ         05h         ; Регистр TrisA - банк 1.
IntCon      equ         0Bh         ; Регистр IntCon.
;.............................................................
;==============================================================================
; Определение названия и положения регистров общего назначения.
;==============================================================================
TimerL      equ         1Bh         ; Регистр младшего разряда 3-байтного
                                    ; двоичного числа.
TimerM      equ         1Ch         ; Регистр среднего разряда 3-байтного
                                    ; двоичного числа.
TimerH      equ         1Dh         ; Регистр старшего разряда 3-байтного
                                    ; двоичного числа.
;.............................................................
;==============================================================================
; Определение места размещения результатов операций.
;==============================================================================
W           equ         0           ; Результат направить в аккумулятор.
F           equ         1           ; Результат направить в регистр.
;==============================================================================
```

```
; Присваивание битам названий.
;=============================================================================
Z           equ         2           ; Флаг нулевого результата.
RP0         equ         5           ; Выбор банка.
;.............................................................
;=============================================================================
            org         0           ; Начать выполнение программы
            goto        START       ; с подпрограммы START.
;*****************************************************************************

;*****************************************************************************
;                           РАБОЧАЯ ЧАСТЬ "ПРОГРАММЫ"
;*****************************************************************************
START       .....................................
;           .....................................
;....       .....................................
;           .....................................
;....       .....................................
;           .....................................
;=============================================================================
; Подготовительные операции.
;=============================================================================
            clrf        IntCon      ; Запрет всех прерываний.
            clrf        Tmr0        ; Сброс в 0 содержимого регистра TMR0.
            clrf        TimerH      ; Сброс в 0 содержимого регистра TimerH.

            bsf         Status,RP0  ; Переход в 1-й банк.
            movlw       b'00100111' ; Предделитель с коэффициентом деления 256
            movwf       OptionR     ; включен перед TMR0, выбор внешнего тактового
                                    ; сигнала с вывода RA4/TOCKI,
                                    ; приращение TMR0 - по переднему фронту.
;=============================================================================
; Начало счета (обозначено меткой On).
;=============================================================================
            movlw       b'00011000' ; Запись в регистр W константы b'00011000'
                                    ; (выводы RA3 и RA4/TOCKI работают "на вход",
                                    ; остальные - "на выход").
On          movwf       TrisA       ; Копирование содержимого регистра W в регистр
                                    ; TrisA (начало счета).
            bcf         Status,RP0  ; Переход в 0-й банк.
;*****************************************************************************
; НАЧАЛО ГРУППЫ ПП ДИНАМИЧЕСКОЙ ИНДИКАЦИИ (вход в малое и большое кольцо
динамической индикации).
;*****************************************************************************
;           .....................................
;....       .....................................
;           .....................................
;....       .....................................
;           .....................................
;=============================================================================
; Первая проверка на переполнение TMR0
;=============================================================================
            btfss       IntCon,2    ; TMR0 переполнен или нет?
            goto        DoNothing   ; Нет, не переполнен --> безусловный переход
                                    ; в ПП DoNothing.
            incf        TimerH,F    ; Да, переполнен --> инкремент содержимого
                                    ; регистра TimerH с сохранением результата
                                    ; инкремента в нем же.
            bcf         IntCon,2    ; Сброс флага переполнения TMR0.
            goto        O_K         ; Безусловный переход в ПП O_K.
DoNothing   nop                     ; Выравнивающий NOP.
            nop                     ; Выравнивающий NOP.
            nop                     ; Выравнивающий NOP.
O_K         .....................................
;           .....................................
;....       .....................................
```

```
;               .....................................
;*******************************************************************************
; НИЖНЯЯ ГРАНИЦА МАЛОГО КОЛЬЦА ДИНАМИЧЕСКОЙ ИНДИКАЦИИ.
;*******************************************************************************
;               .....................................
;*******************************************************************************
; НИЖНЯЯ ГРАНИЦА БОЛЬШОГО КОЛЬЦА ДИНАМИЧЕСКОЙ ИНДИКАЦИИ.
;*******************************************************************************
; Группа команд точной "доводки" величины интервала времени основного счета
; до расчетного значения.
;===============================================================================
;               .....................................
;               .....................................
;===============================================================================
; Конец счета (обозначен меткой Off).
;===============================================================================
                clrw                    ; Сброс в 0 содержимого регистра W.
                movwf   PortA           ; Копирование нуля из регистра W
                                        ; в регистр PortA.
                bsf     Status,RP0      ; Переход в 1-й банк.
                movlw   b'00010000'     ; Запись в регистр W константы b'00010000'
                                        ; (вывод RA4/TOCKI работает "на вход",
                                        ; остальные - "на выход").
Off             movwf   TrisA           ; Копирование содержимого регистра W в регистр
                                        ; TrisA (конец счета).
                bcf     Status,RP0      ; Переход в 0-й банк.
;===============================================================================
; Вторая проверка на переполнение TMR0
;===============================================================================
                btfss   IntCon,2        ; TMR0 переполнен или нет?
                goto    Analyse         ; Нет, не переполнен --> безусловный переход
                                        ; в ПП Analyse.
                incf    TimerH,F        ; Да, переполнен --> инкремент содержимого
                                        ; регистра TimerH, с сохранением результата
                                        ; инкремента в нем же.
                bcf     IntCon,2        ; Сброс флага переполнения TMR0.
;===============================================================================
; Копирование содержимого регистра TMR0 в регистр TimerM.
;===============================================================================
Analyse         movf    Tmr0,W          ; Копирование содержимого регистра TMR0
                                        ; в регистр W.
                movwf   TimerM          ; Копирование содержимого регистра W
                                        ; в регистр TimerM.
;===============================================================================
; Сброс в 0 содержимого регистра TimerL.
;===============================================================================
                clrf    TimerL          ; Сброс в 0 содержимого регистра TimerL.
;===============================================================================
; Подпрограмма досчета.
;===============================================================================
CountIt         incf    TimerL,F        ; Инкремент содержимого регистра TimerL с
                                        ; сохранением результата инкремента в нем же.
;-------------------------------------------------------------------------------
; Кратковременная разблокировка счетного входа TMR0.
;-------------------------------------------------------------------------------
                bsf     PortA,3         ; Формирование на выводе RA3 единицы.
                nop                     ; Временной "зазор".
                bcf     PortA,3         ; Формирование на выводе RA3 нуля.
                nop                     ; Временной "зазор".
;-------------------------------------------------------------------------------
; Досчет.
;-------------------------------------------------------------------------------
                movf    Tmr0,W          ; Копирование содержимого регистра TMR0
                                        ; в регистр W.
                bcf     Status,Z        ; Сброс флага нулевого результата Z.
                subwf   TimerM,W        ; Вычесть из содержимого регистра TimerM
```

```
                                ; содержимое регистра W.
            btfsc       Status,Z    ; Результат операции вычитания равен
                                ; или нет нулю?
            goto        CountIt     ; Да, равен ---> переход в ПП CountIt.
            comf        TimerL,F    ; Нет, не равен ---> инвертировать все биты
                                ; регистра TimerL, с сохранением результата
                                ; инвертирования в нем же.
            incf        TimerL,F    ; Инкремент содержимого регистра TimerL, с
                                ; сохранением результата инкремента в нем же.
;===============================================================================
; На данный момент, в 3-байтном регистре TimerH/TimerM/TimerL сформирован
; результат подсчета в виде 3-байтного двоичного числа, которое, далее, можно
; обрабатывать в группе подпрограмм преобразования двоичных чисел в двоично-
; десятичные или в группах подпрограмм другого предназначения (в зависимости от
; специфики разрабатываемого устройства).
;           .....................................
;           .....................................
;....       .....................................
;           .....................................
;....       .....................................
;           .....................................
;*******************************************************************************
            end                 ; Конец программы.
```

Разбираем текст программы **Tmr0.asm**.
Группа команд счета начинается с команд подготовительных операций, сразу же после которых следуют команды начала счета.
Группа команд подготовительных операций состоит из команды запрета прерываний (**clrf IntCon**), команды сброса в 0 содержимого регистра **TMR0** (**clrf Tmr0**), команды сброса в 0 содержимого регистра **TimerH** (**clrf TimerH**) и двух команд записи константы в регистр специального назначения **OPTION**.
Регистр **OPTION** "лежит" в 1-м банке, следовательно, перед работой с ним, нужно перейти в 1-й банк (**bsf Status,RP0**).
После исполнения команды **movwf Option**,
- вывод **RA4/TOCKI** подключится к входу предделителя, а выход предделителя подключится к входу **TMR0**,
- коэффициент деления предделителя будет равен **256**,
- приращение содержимого **TMR0** будет происходить по перепаду от **0** к **1**.

Есть 2 способа обозначения перепадов импульса.
- смена* 0 *на* 1 → *передний фронт импульса, смена* 1 *на* 0 → *задний фронт импульса.
- смена* 0 *на* 1 → *фронт импульса, смена* 1 *на* 0 → *спад импульса.
После исполнения команд подготовительных операций, начинается формирование интервала времени счета.
Для этого необходимо, чтобы вывод **RA4/TOCKI** (счетный вход ПИКа / вход предделителя) и вывод блокировки **RA3** работали "на вход".
В этом случае, сопротивление между выводом **RA3** и корпусом велико (разблокировка счета).
Регистр **TrisA** "лежит" также в 1-м банке, поэтому банк менять не нужно (1-й банк был установлен ранее).
Константа **00011000**, через регистр **W**, "переправляется" в регистр **TrisA**, после чего, выводы **RA3** и **RA4/TOCKI** настраиваются на работу "на вход", а все остальные выводы порта А, на работу "на выход".
Так как далее будут производиться операции с регистрами 0-го банка, в конце группы команд начала счета, необходимо "вернуться" в 0-й банк (**bcf Status,RP0**).
Убедитесь, что в тексте программы **Tmr0.asm**, эта последовательность действий соблюдается.
Счет разрешается после того, как константа будет записана в регистр **TrisA**, то есть, после исполнения команды **movwf TrisA**.
В тексте программы, для удобства, я пометил эту команду меткой **On** (начало основного счета).
Если эта метка не нужна, то ее можно убрать, так как в тексте программы, обращений к ней нет.

То же самое относится и к метке **Off**, которой помечен конца основного счета.
Примечание: установка метки (меток), к которой нет обращения, не является ошибкой, и работа программы от этого не нарушится.
Ошибкой является обращение к метке, которой нет (которая не установлена).
Итак, перед входом рабочей точки программы в ПП динамической индикации, произведены подготовительные операции и разрешен счет.
В соответствии со сказанным выше, далее, должна быть произведена первая проверка на переполнение **TMR0**.
Она должна "врезана" в малое кольцо динамической индикации.
В этом случае, обеспечивается периодичность проверок, гарантирующая безошибочный подсчет количества переполнений **TMR0** (см. сказанное ранее).
Первая проверка на переполнение **TMR0** начинается с опроса состояния флага прерывания по переполнению **TMR0** (**btfss IntCon,2**), поднятие которого не зависит от того, разрешены прерывания или нет, а зависит только от факта переполнения **TMR0**.
Ранее, в подготовительных операциях, все биты регистра **IntCon** были сброшены в **0**, следовательно, на момент разрешения счета, флаг прерывания по переполнению **TMR0** (**T0IF**) будет опущен.
Напоминаю, что флаг **T0IF** это флаг 2-й группы, и если он поднялся, то опускать его нужно программно.
Команда ветвления **btfss IntCon,2** "разветвляет" программу на 2 сценария, которые, ниже по тексту программы, снова сходятся на первой команде ПП с условным названием **O_K** (или на команде, помеченной меткой **O_K**, если это считать меткой).
В случае наличия подобного рода "разветвления" на 2 сценария, с последующим их "схождением" (а это и имеет место быть), необходимо принять меры по "затяжке" времени исполнения сценария с меньшим временем исполнения (выравнивание).
То есть, интервал времени исполнения этого сценария нужно сделать в точности равным интервалу времени исполнения другого сценария (о такой необходимости говорилось ранее).
Оба этих сценария исполняются внутри такого "важняка", как малое кольцо динамической индикации, и поэтому они сильно влияют на процесс формирования калиброванного интервала времени счета.
Следовательно, выравнивание необходимо. Что Вы и видите в тексте программы (см. выравнивающие **NOP**ы).
Разбираем сценарии.

1-й сценарий.

Если произошло переполнение **TMR0** (флаг **T0IF** поднялся), то команда **goto DoNothing** не исполняется (вместо нее исполняется "виртуальный" **NOP**), и рабочая точка программы "встает" на команду **incf TimerH,F**.
После исполнения этой команды, происходит инкремент содержимого регистра **TimerH**, с сохранением результата инкремента в нем же.
Так как в ходе отработки ПП динамической индикации, формируется большое количество малых колец динамической индикации, то будет происходить и такое же большое количество проверок на переполнение **TMR0**.
В любом из случаев обнаружения факта поднятия флага **T0IF**, будет осуществлен инкремент содержимого регистра **TimerH**.
Таким образом, к моменту выхода рабочей точки программы из большого кольца динамической индикации, в регистре **TimerH** "осядет" число, равное количеству переполнений **TMR0**, произошедших с момента разрешения основного счета и до момента выхода рабочей точки программы из большого кольца динамической индикации.
Это "осевшее" число находится в прямо пропорциональной зависимости от частоты импульсной последовательности, подаваемой на счетный вход ПИКа.
После инкремента содержимого регистра **TimerH**, для обеспечения необходимого условия осуществления следующей проверки (флаг **T0IF** должен быть опущен), с помощью команды **bcf IntCon,2**, бит флага **T0IF** сбрасывается в **0**.
После этого происходит безусловный переход в ПП **O_K**.

2-й сценарий.

Если переполнения **TMR0** не произошло (флаг **T0IF** не поднялся), то команда инкремента содержимого регистра **TimerH** не выполняется, а происходит безусловный переход в ПП "выравнивания" с названием **DoNothing** (**goto DoNothing**).
О "выравнивающих" **NOP**ах говорилось ранее.

Почему именно три **NOP**а?
Давайте разбираться.
Если считать "точкой разветвления", на 2 сценария, команду ветвления **btfss IntCon,2**, а "точкой их слияния" первую команду ПП **O_K** (а это так и есть), то исполнение самого короткого из этих двух сценариев (**btfss** - 1 м.ц. / **goto** - 2 м.ц.) займет **3 м.ц.**, а исполнение самого длинного (**btfss** - 1 м.ц. / "виртуальный" **NOP** - 1 м.ц. / **incf** - 1 м.ц. / **bcf** - 1 м.ц. / **goto** - 2 м.ц.) займет **6 м.ц.**.
6-3=3.
Следовательно, для "подтягивания" времени исполнения короткого сценария, к времени исполнения длинного сценария, необходимо **3** "выравнивающих" **NOP**а, что Вы и видите в тексте программы.
Соответственно, ПП **DoNothing** должна исполняться только при отработке короткого сценария.
После группы команд 1-й проверки на переполнение **TMR0**, следуют остальные команды ПП динамической индикации и группа команд точной "доводки" интервала времени основного счета до рассчетного значения.
После отработки цикла динамической индикации и исполнения группы команд точной "доводки" интервала времени счета до расчетного значения, необходимо завершить счет.
Перед тем как это сделать, необходимо, на выходе защелки вывода **RA3**, установить нулевой (блокирующий) уровень.
В данном случае, в **0** сбрасывается не один только бит **№3** регистра **PortA** (можно сделать и так: **bcf PortA,3**), а весь байт.
Причем, эта операция организована не с использованием команды **movlw 0**
(плюс, **movwf PortA**), а с использованием команды сброса в **0** содержимого регистра **W** (**clrw**).
Посмотрите как это сделано в тексте "программы" (**clrw** и **movwf PortA**).
Это можно считать примером использования команды **clrw**.
Установка, на выходе защелки **№3** порта А, нулевого уровня не есть окончание счета, так как вывод **RA3** работает "на вход" (выход защелки отключен от вывода **RA3**).
Счет заканчивается тогда, когда этот вывод переключится с работы "на вход" на работу "на выход".
Команда, после исполнения которой счет прекращается, помечена меткой **Off**.
Так как в процессе выхода рабочей точки программы из ПП динамической индикации (на "участке" последнего "витка" большого кольца динамической индикации, который отрабатывается после 1-й проверки) и дальнейшего исполнения группы команд точной "доводки" интервала времени счета до расчетного значения, может произойти переполнение **TMR0**, то необходимо организовать вторую проверку на переполнение **TMR0**.
Принцип ее организации такой же, как и первой проверки.
В этом случае (проверка №2), нет необходимости в выравнивании сценариев, так как проверка производится тогда, когда основной счет закончен.
Если исполняется один сценарий (переполнения **TMR0** нет), то инкремента содержимого регистра **TimerH** не происходит и осуществляется безусловный переход в ПП **Analyse**.
Если исполняется другой сценарий (переполнение **TMR0** есть), то происходит инкремент содержимого регистра **TimerH**, флаг **T0IF** программно опускается и начинается отработка всё той же ПП **Analyse** ("все пути ведут в Рим").
В ходе 2-й проверки на переполнение **TMR0**, количество инкрементов не может быть больше одного, так как после первого же инкремента, рабочая точка программы выходит из проверки №2.
После окончания этой проверки, содержимое регистра **TimerH** будет окончательно сформировано и можно перейти к формированию содержимого регистра **TimerM**.
В этом случае, все просто: содержимое регистра **TMR0** копируется (через регистр **W**) в регистр **TimerM** (**movf Tmr0,W** и **movwf TimerM**).
После этого, в регистре **TimerM**, будет "лежать" содержимое **TMR0** на момент окончания основного счета.
Вопрос: "Почему, перед проведением столь ответственной операции с содержимым регистра **TimerM**, его содержимое предварительно не было сброшено в **0**, как например, перед проведением не менее ответственной операций с содержимым регистра **TimerH**"?
Ответ: А зачем? Ведь происходит запись "по верху" (то, что "лежало до того", не имеет значения).

Что касается содержимого регистра **TimerL** на момент начала ПП досчета **CountIt**, то его, перед началом этой ПП, обязательно нужно сбросить в **0**, так как далее будут производиться операции с "привязкой к начальной точке отсчета" (счет от нуля).
Что и сделано (**clrf TimerL**).
Теперь можно перейти к формированию содержимого регистра **TimerL**.

Работа подпрограммы досчета

Итак, функция ПП досчета (**CountIt**) заключается в "выталкивании" числа, зафиксировавшегося в байте предделителя (на момент окончания основного счета), в регистр **TimerL**.
То есть, ПП досчета **CountIt** является специфической ("изощренной") разновидностью копирования этого числа в регистр общего назначения.
Это копирование происходит не напрямую, а косвенно, то есть, посредством организации "принудиловки", смысл которой "заложен" в той "математике", которая описана выше.
ПП досчета **CountIt** начинается с инкремента содержимого регистра **TimerL**, с сохранением результата инкремента в нем же (**incf TimerL,F**).
Далее, путем смены уровней на выходе защелки **№3** порта А (вывод **RA3**), формируется короткий счетный импульс.
Ничего сложного в процессе его формирования нет (см. комментарии в тексте программы).
По переднему фронту этого импульса, происходит приращение (инкремент) числового значения байта предделителя.
Далее, с целью ответа на вопрос: "Изменилось или не изменилось числовое значение байта регистра **TMR0**?", организуется проверка.
Для осуществления этой проверки, необходимо произвести сравнение числа, "лежащего" в регистре **TMR0** до прохождения счетного импульса, с числом, "лежащим" в регистре **TMR0** после прохождения счетного импульса (отслеживание факта приращения).
Ранее, число, "осевшее" в регистре **TMR0** на момент окончания счета, было скопировано из регистра **TMR0** в регистр **TimerM** (см. ПП **Analyse**).
Это "эталон" (он не изменяется), с которым будет производиться сравнение.
С этим "эталоном" будет сравниваться содержимое регистра **TMR0**, которое, в итоге отработки ПП **CountIt**, неизбежно увеличится на **1**.
Это произойдет в случае смены числового значения байта предделителя с **.255** на **.00**, после чего будет осуществлено числовое преобразование, и рабочая точка программы выйдет из ПП **CountIt** по сценарию "программа исполняется далее".
Во всех остальных случаях, инкремента содержимого регистра **TMR0** (возможен только один инкремент) производится не будет (будут производиться инкременты содержимого байта предделителя), и рабочая точка программы будет "мотать витки" в ПП **CountIt**.
На каждом таком "витке", формируется один счетный импульс (см. ПП **CountIt**), который инкрементирует содержимое регистра **TimerL**.
В процессе "намотки этих витков", инкременты содержимого регистра **TimerL** будут происходить до тех пор, пока числовое значение байта предделителя не сменится с **.255** на **.00**.
В момент этой смены, по определению, происходит инкремент содержимого регистра **TMR0**.
Это приведет к тому, что в ходе следующей проверки, будет обнаружено числовое расхождение между "эталоном" (числовым значением байта регистра **TimerM**) и числовым значением байта регистра **TMR0**.
После этого, будет осуществлено числовое преобразование, и рабочая точка программы выйдет из ПП **CountIt** по сценарию "программа исполняется далее".
На момент начала исполнения этого сценария, числовое значение байта регистра **TimerL** будет в точности равно числовому значению байта предделителя на момент окончания основного счета.
"Мотание колец" происходит в группе команд, реализующих специфическую, циклическую ПП задержки.
Величина этой задержки зависит от числового значения байта предделителя на момент окончания основного счета.
Чем бОльшим будет это значение, тем меньшей будет задержка (и наоборот).
Эту группу команд можно классифицировать как группу команд, реализующих ***однобайтный, суммирующий счетчик с "плавающей" предустановкой***.

То есть, от "витка к витку" полного цикла программы, числовое значение предустановки может быть различным.
Этот счетчик как бы работает в режиме "автономного самообслуживания".
То есть, досчет не влияет на результаты подсчета, произведенного ранее (в ходе процедуры досчета, не вносится погрешностей в результат основного счета).
Процедура досчета является чисто технической процедурой "переправки" числа из байта предделителя в байт регистра **TimerL**.
Вернемся к тексту ПП досчета.
После формирования счетного импульса, исполняется команда **movf Tmr0,W**.
То есть, содержимое регистра **TMR0** копируется в регистр **W**.
Тем самым, обеспечивается дальнейшая возможность сравнения числа, записанного в регистре **TMR0**, с "эталоном".
Для обеспечения комфортности отслеживания работы подпрограммы, флаг нулевого результата **Z** предварительно сбрасывается в **0**: **bcf Status,Z** (если эта комфортность не нужна, то команду **bcf Status,Z** можно "выкинуть" из текста программы).
Далее, при помощи команды **subwf TimerM,W**, производится сравнение чисел, записанных в байтах регистров **W** и **TimerM**.
Сравнение производится путем вычитания содержимого регистра **W**, из содержимого регистра **TimerM**, с последующей проверкой на нулевой результат.
В регистре **TimerM** находится "эталон", то есть, число, "зафиксировавшееся" в **TMR0** после окончания счета.
В регистре **W**, может находиться либо число равное "эталону" (если переполнения байта предделителя нет), либо число отличное от "эталона" (если переполнение байта предделителя есть).
Если после исполнения команды **subwf TimerM,W** (команда **subwf** воздействует на флаг **Z**), произвести проверку состояния флага **Z** (**btfsc Status,Z**), что в тексте ПП **CountIt** и сделано, то в первом случае, будет исполнен сценарий "намотки витков", а во втором случае, сценарий "программа исполняется далее".
В сценарии "намотки витков", по команде **goto CountIt**, происходит переход рабочей точки программы на начало исполнения ПП досчета, после чего, ПП досчета исполняется снова, и содержимое регистра **TimerL** еще раз инкрементируется.
И так далее.
Такие "витки" будут "наматываться" до тех пор, пока числовое значение байта предделителя не изменится с **.255** на **.00** (переполнение), после чего произойдет инкремент содержимого регистра **TMR0**.
На этот момент времени, содержимое регистра **TimerL** будет инкрементировано количество раз, равное разнице между числом 256 и числом, зафиксировавшимся в байте предделителя на момент окончания основного счета.
После инкремента содержимого **TMR0**, результат исполнения команды **subwf TimerM,W**, станет не =**0**, и произойдет переход на сценарий "программа исполняется далее".
После этого, рабочая точка программы "зайдет" в группу команд числового преобразования.
Далее все очень просто: вспомните, как, занимаясь математикой, мы "химичили" с числом **.192** (см. выше).
То есть, последовательность действий должна быть следующей:

1. Инверсия всех битов содержимого регистра **TimerL**, с сохранением результата инверсии в нем же (**comf TimerL,F**).
2. Инкремент содержимого регистра **TimerL**, с сохранением результата инкремента в нем же (**incf TimerL,F**).

После этого, рабочая точка программы, по сценарию "программа исполняется далее", выходит из ПП досчета и "дальше бежит делать свои дела".

Вопрос: "Можно ли обойтись без второй проверки на переполнение **TMR0**"?
Ответ: можно, если сразу же после первой проверки на переполнение **TMR0**, счет прекратить.
Если организуются 2 проверки на переполнение **TMR0**, то время прохождения рабочей точкой программы участка программы, между окончанием проверки №1 и началом проверки №2, не должно превышать времени двойного переполнения **TMR0** на самой высокой, расчетной скорости счета, а иначе, 2 переполнения посчитаются как одно.
При выборе частоты проверок на переполнение **TMR0** № 1, следует учитывать то, что чем

чаще они следуют, тем бОльшую программную скорость счета можно достигнуть, но и слишком частыми их делать не стОит, так как превышение программной скорости счета над предельно допустимой скоростью счета применяемого ПИКа, в практическом отношении, выигрыша не даст (выше того, на что "способен" счетный вход ПИКа, не "прыгнешь").
Процесс счета происходит в границах интервала времени счета, и именно в этих границах, должны быть организованы проверки на переполнение **TMR0**.
Это предполагает наличие, в этих границах, циклической подпрограммы, в которую и нужно "врезать" группу команд проверки №1 на переполнение **TMR0**.
Количество "витков, наматываемых" в ней рабочей точкой программы, в интервале времени между ее "влётом" в нее и "вылетом" из нее, есть количество проверок (№1) на переполнение **TMR0**, плюс, одна проверка №2.
Если речь идет о "влёте" в циклическую ПП, с последующим "вылетом" из нее, то это предполагает наличие счетчика внутренних циклов ("постановка циклов на счетчик").
В конечном итоге, задача сводится к недопущению переполнения **TimerH** во всем диапазоне частот работы ПИКа и к подбору оптимального времени отработки одного цикла ПП **CYCLE**.
Этот интервал времени должен быть стабилен (напоминаю про выравнивание сценариев).
"Врезать" что-то дополнительное, в "промежуток" от начала и до конца счета, можно, но нужно учитывать то, что после этого, нужно производить калибровку.
В следующем разделе, я расскажу, как работает "полноценная" программа частотомера.

.....
.....
.....

То, что Вы прочитали, написано в довольно-таки сдержанном тоне.
Далее, постепенно, будет "вводиться в эксплуатацию" образное мышление.
Появятся различные "персонажи".
То, что жизнь нужно разнообразить, понятно, но дело не только в этом.
Речь идет о "вводе в эксплуатацию" такого мощнейшего "инструмента", как подсознание.
Многие не осознают его мощи. А зря …

Заключение

Уважаемые читатели.

Работа над "Самоучителем..." функционально закончена.
Я объяснил работу всех составных частей "начинки" PIC16F84A, основы конструирования устройств на м/контроллерах и "провел" Вас по всей "цепочке" от возникновения нескольких идей и до их практической реализации, с постепенным наращиванием сложности решаемых задач.
Это чисто авторская информация, и она является редкой в том смысле, что насколько я знаю, никто кроме меня так и не отважился "броситься на эту амбразуру" (подробное описание процесса конструирования достаточно сложного устройства на м/контроллере, с предъявлением к этому описанию высоких требований по "детализации" и "удобоваримости").
Покажите мне этого человека, и я буду уважать его до конца жизни, так как знаю, какой это большой труд.
Я постарался обеспечить Вам доступ в "святая святых" процесса конструирования устройств на м/контроллерах, то есть туда, где на "входе стоит огромный шлагбаум, и его сторожит куча свирепых псов".
Детальное описание этой "ужасной картины" вовсе не есть процесс легкий и приятный, но "врага нужно знать в лицо", а иначе с собственным лицом будут большие проблемы.
Без хорошего "пулемета и гранаты", будет та же ситуация, что и с Матросовым: чести много, а толку, лично для него, никакого.
Гораздо прагматичнее и разумнее, с комфортом, спокойно и по-деловому, "перестрелять этих псов из пулемета и шлагбаум взорвать".
Вот это уже деловой подход сильного и уважающего себя мужика (в самом лучшем смысле этого слова), который, лично мне, больше по душе.
На эту работу я потратил почти 2 года "плотной" работы.
При этом, я стремился вложить в нее свою душу, сделать информацию максимально понятной и создать "климат" доброжелательности и дружелюбия, основанный на взаимном уважении к труду друг друга.
В любом случае, за свою работу мне не стыдно, так как я делал ее с удовольствием и добросовестно.
Огромное, человеческое спасибо всем тем, кто меня, в том или ином виде, поддерживал и поддерживает в работе.
Без этого, я не смог бы осилить эту "глыбу" информации.
Если говорить откровенно, то в течение некоторого времени, я считал себя "лохом".
В том смысле, что "выдаю на гора" такую информацию, которую "днем с огнем не сыщешь", и делаю ту работу, за которую, ввиду ее высокой трудоемкости, никто не берется.
И переубедили меня в этом Вы.
Если работа востребована, то все видится в другом свете.
В основном, трудности "въезда в м/контроллеры" связаны с отсутствием понятной и доступной, русскоязычной "школы", способной "пробить мозги до самого гипофиза".
Имеющиеся в наличии курсы обучения, в том числе и в ВУЗах, крайне неэффективны, так как они представляют собой форму без осмысленного и понятного содержания, да к тому же еще и не очень-то "привязанную" к реальным, жизненным потребностям.
Я это знаю совершенно точно, так как общаюсь со студентами ВУЗов.
Дело доходит даже до того, что преподаватели сами признают неэффективность того, что они преподают.
Вокруг этой "железяки" с названием м/контроллер, из-за нашего страха, природной лени и бездумного "поклонения идолу", создан такой ажиотаж (о, этот "великий и ужасный" м/контроллер!!!), что просто диву даешься.
Нет в нем ничего "великого и ужасного". Это же "тупая железяка" !!!
Что Вы ей приказали, то она и сделает.
Проблема только в том, как научиться толково и грамотно приказывать.
Но из-за отсутствия продуманной и эффективной системы обучения, на "фоне" полнейшей анархии (кто во что горазд), попытки этому научиться, во многих случаях, заканчиваются тем, что человек начинает чувствовать себя дебилом (условно).
Это не "осчастливливает", а "отбивает почки".
Все это я капитально прочувствовал еще до того, как затеял свою работу (имею огромный

опыт "вождения носом по батарее" и "битья головой о стену", что кстати, не есть однозначно плохо).
Но кто-то же ведь должен положить конец этому "безобразию" или, по крайней мере, попытаться это сделать?
А там, глядишь, и дело с мертвой точки сдвинется...
Обращаю Ваше внимание на то, что конструирование устройств на м/контроллерах (и вообще конструирование) это искусство, то есть, удел людей творческих и увлеченных.
Если вы относитесь к этой категории "гусаров-схимников", то "милости просим к нашему шалашу" и давайте "кучковаться". Так выжить легче.
Свою главную задачу я вижу в том, чтобы внести свой посильный вклад в работу по созданию качественной информации, реально помогающей людям.
"Самоучитель..." можно рассматривать как первый этап этой помощи.
Образно выражаясь, те из Вас, кто его изучил и понял, о чем шла речь, уже являются, хотя и "зелеными", но "лейтенантами".
Можете смело праздновать "выпускной бал" и любоваться "виртуальной корочкой" об окончании серьезного "ВУЗа", которая и есть "пропуск" в интересный, увлекательный и своеобразный мир м/контроллеров.

Успехов Вам в работе!
С уважением, Корабельников Евгений Александрович (родом из СССР).

Дополнительно

1. О задержках программы Multi.asm.

При применении кварца на 4 мГц, максимально возможная частота мультивибратора составляет 50 кГц.
Если нужно получить частоту выше чем 50 кГц, то нужно применить кварц с частотой более 4 мГц.
Если нужно получить частоту порядка нескольких герц или долей герца (или еще ниже), то нужно применить не однобайтный, а 2-х или 3-хбайтный счетчик.
Для решения большинства задач, связанных с формированием низкочастотных сигналов, обычно, достаточно и 2-хбайтного счетчика, работа которого описана при "разборках" с программой **cus.asm**.
N однобайтных счетчиков (задержки отрабатываются последовательно)**, не есть N-байтный счетчик.**
Например, в случае применения 2-хбайтного счетчика, обеспечивается гораздо бОльшая задержка, чем в случае последовательной отработки двух задержек, формируемых однобайтными счетчиками.

2. Информация по работе с портами.

Не стОит пытаться управлять защелками выводов портов из "шапки" программы.
Из этого ничего путного не получится.
После безошибочной "прописки" регистров **TrisA**, **PortA**, **TrisB**, **PortB** (а также и других регистров специального назначения) по адресам, указанным в распечатке области оперативной памяти, про эти строки (в "шапке" программы) можно вообще "забыть" (условно).
Управление защелками портов осуществляется только в рабочей части программы.
Если нужно изменить состояние только одной из защелок, то используются бит-ориентированные команды **bcf** или **bsf**.
Если нужно изменить состояния сразу нескольких защелок (или вообще всех), то используются байт-ориентированные команды.
Например, необходимо установить на выводах
RB0, RB1, RB2, RB3, RB4, RB5, RB6, RB7: 1, 0, 0, 1, 1, 0, 0, 0 соответственно.
Для реализации этого действия, в регистр **PortB**, нужно записать число **00011001**.
Непосредственно, записать константу, в регистр **PortB**, нельзя.
Это можно сделать только через регистр **W**.
Делается это так:

```
movlw     b'00011001'
movwf     PortB
```

Вместо **b'00011001'**, можно использовать число **.25** или **19h**.
К соответствующим выводам портов, выходы защелок подключаются только тогда, когда эти выводы настроены на работу "на выход".
От выводов портов, настроенных на работу "на вход", выходы соответствующих защелок отключены.
Переключениями направлений работы выводов портов "рулят" биты регистров **TrisA** и **TrisB**.
Вывод из этого следующий***: программа "способна" управлять состояниями*** (0 или 1) ***только тех выводов портов, которые настроены на работу "на выход", а состояния выводов портов, настроенных на работу "на вход", будут определяться внешними источниками сигналов.***
Если вернуться к приведенному выше примеру и предположить, что например, выводы **RB0, RB1, RB2, RB3** настроены на работу **на выход**, а выводы **RB4, RB5, RB6, RB7** настроены на работу **на вход**, то после исполнения указанных выше двух команд, на выводах **RB0, RB1, RB2, RB3** установятся **1, 0, 0, 1** соответственно, а состояния выводов **RB4, RB5, RB6, RB7** будут определяться внешними источниками сигналов, не смотря на то, что на выходах защелок этих выводов установятся **1, 0, 0, 0** соответственно.
Что касается последнего, то, по большому счету, толку от этой установки – никакого.

Так же, как и от установки любой комбинации нулей и единиц, ведь выходы этих защелок не подключены к выводам порта B.

3. О надежности работы программы.

Вопрос по программе **Multi.asm**: "Какой смысл включать первые 4 команды ПП **Start** в полный цикл программы, ведь исполнив их один раз (на 1-м "витке"), далее, их можно не исполнять"?
То есть, речь идет о выставлении метки на команде **movlw .32** и о замене команды **goto Start** (в рабочей части программы) на команду **goto *название метки***?
Представьте себе достаточно обычную ситуацию: плохой контакт с батарейкой или по сети 220v "прошла" импульсная помеха в то время, когда мультивибратор работает.
Если при этом произойдет "несанкционированное" изменение содержимого регистров **Status** и/или **TrisB**, то устройство может перестать работать (простейший случай зависания программы).
В этом случае, восстановить работоспособность устройства можно только после выключения и последующего включения питания.
Если же безусловный переход осуществляется в ПП **Start**, то в случае наличия "бяки", последствия этого сбоя будут устранен на следующем "витке" полного цикла программы, без манипуляций с выключателем питания.
То есть, речь идет о "перестраховке", необходимость которой определяет программист.
Она вовсе не обязательна, но знать об этом нужно ("в жизни всякое бывает").

4. О директиве EQU.

При присвоении битам регистров специального назначения их названий (в "шапке" программы), названия битов повторяться не должны, но номера битов повторяться могут.
Например, биту **№0** можно одновременно присвоить названия **C** и **RBIF**
(**C equ 0** и **RBIF equ 0**).
В этом случае, если рабочая часть команды → **Status,C**, то команда обратится к биту **№0** регистра **Status**, а если рабочая часть команды → **IntCon,RBIF**, то команда обратится к биту **№0** регистра **IntCon**.

5. О регистре OptionR.

В "шапках" текстов программ, а следовательно и в рабочих частях программ, название регистра специального назначения **OPTION** обязательно должно иметь в своем названии букву **R**.
Я пишу так: **OptionR**.
Если название этого регистра не будет содержать буквы **R**, то после ассемблирования текста программы, **MPLAB** выдаст сообщение об ошибке и **HEX-файл** создан не будет (можете проверить).

6. Чем отличается PIC16F84 от PIC16F84A.

По "цоколевке", они полные аналоги.
Есть отличия в электрических и временнЫх характеристиках, но они не очень существенны.
С точки зрения обеспечения лучших временнЫх характеристик, более предпочтителен **PIC16F84A**.
С точки зрения обеспечения лучших электрических характеристик, более предпочтителен **PIC16F84**.
Так как основная часть устройств, собранных на этих м/контроллерах, работает в электрических режимах, которые далеки от предельно допустимых, то в этом случае, предпочтение нужно отдавать **PIC16F84A**.
Сравнительная таблица электрических и временных характеристик этих ПИКов "лежит" на странице 78 даташита **PIC16F84A**.

7. Каков верхний предел быстродействия PIC16F84A по его счетному входу (вывод RA4/TOCKI)?

"Перестраховочный" вариант разработчиков - 30 Мгц., но реально, он выше.
Вполне обычным является "потолок" 50 … 60 Мгц., а для некоторых "выдающихся экземпляров", и выше. Но на это уповать не стОит.
Короче, кому как повезет.

8. О "плавающей" запятой.

Практическая необходимость в организации "плавающей" запятой появляется при наличии нескольких пределов измерений (подсчетов).
Если имеется только один предел измерения, как например, в программе 7-разрядного, "чистого" частотомера **Kea.asm**, то необходимости, в организации "плавающей" запятой, нет.
Проще говоря, "проволочину", подключенную к 13-му выводу ПИКа (**RB7**), можно отключить, после чего, через гасящий резистор (примерно 470 ом.), подключить этот вывод к +5в. (запятая постоянно включена), и группу команд установки запятой можно аннулировать.
При этом освобождается вывод порта В **RB7**, который можно использовать для чего-то другого.
Номинал гасящего резистора запятой нужно подобрать таким образом, чтобы яркость ее свечения (она будет светиться ярко) была примерно такой же, как и яркость свечения остальных сегментов.
Естественно, что после такого изменения текста программы (вмешательства в "святая святых"), необходимо произвести коррекцию интервала времени измерения в сторону его увеличения.
Примечание: в текст программы **Kea.asm**, группа команд установки запятой введена в обучающих целях.
Для того чтобы организовать "плавающую" запятую (речь идет о программном изменении положения запятой, в зависимости от выставленного предела измерения), в текст программы **Kea.asm**, нужно ввести группу команд опроса клавиатуры и назначить дополнительный регистр общего назначения. Например, **Pin**.
В зависимости от результата опроса клавиатуры, в него будут записываться соответствующие константы (эту запись можно произвести ниже точки входа во 2-й и последующие "витки" полного цикла программы), которые и будут определять положения запятой.
А дальше все просто: в группе команд установки запятой, команда **movlw 5** заменяется на команду **movfw Pin**.
Естественно, что количество положений запятой может быть и более двух.
Это зависит от замысла программы.
В дальнейшем, я в деталях расскажу, как именно организуется работа с "плавающей" запятой.

9. Пояснения к программам Retr_1.asm и Retr_3.asm.

Для программы **Retr_1.asm**.
В данном случае, **WDT** (включен в битах конфигурации) работает без предделителя (предделитель включен перед **TMR0**, который не задействуется, и следовательно, предделитель отключен от **WDT**) и имеет время срабатывания примерно **18 мс.**
Учитывая то, что по ходу исполнения программы, он сбрасывается очень часто, этого времени с избытком хватает для обеспечения "безсбросной" отработки программы.
То есть, в данном случае, необходимости в подключении предделителя, к **WDT**, нет, и не нужно "ломать голову" над заданием его коэффициента деления.
В битах **№ 0,1,2**, можно выставить **0** (а можно и единицы или комбинацию нулей и единиц. Это не влияет на работу программы), плюс **0** в бите **№3** (предделитель включен перед **TMR0**, а следовательно, отключен от **WDT**), плюс **0** в битах **№4** и **5** (не имеет значения, можно выставить и 1), плюс **0** в битах **№6** и **7** (определено заданием на разработку).
Вот и получается, что в регистр **OptionR** можно записать константу **.00**.
Для программы **Retr_3.asm**.
Кроме сказанного выше, значение бита **№6** не важно (уходов в прерывания нет).
Поэтому в комментарии и сказано "остальное не существенно", но в битах **№3** и **7** должны быть установлены нули.
В остальных битах, можно выставить "все что угодно" (не важно), но проще всего выставить нули. Что и сделано.

10. Как "загрузить" текст программы, в текстовый редактор MPLAB, из "Винворда" (расширение .doc) и других текстовых редакторов?

Например, текст программы находится в файле с расширением **.doc**.
В "Винворде", открываете этот файл, щелкаете по ***Выделить все***, а затем, по ***Копировать***.
После этого, текст программы "уйдет" в буфер обмена.

Закрываете "Винворд" (появляется диалоговое окно с вопросом ***Нужно ли сделать содержимое буфера обмена доступным другим приложениям, установленным на Вашем компьютере?*** Жмите ***Да***).
Создаете проект (описано в "Самоучителе...").
Щелкаете по ***Вставить***, и текст программы скопируется в пустое окно созданного Вами ASM-файла.
Вернее всего, после этого, форматирование текста будет нарушено, и после ассемблирования, будет выдана ошибка **Error 8** (HEX-файл создан не будет).
Если это так, то в соответствии с "правилом 12-ти пробелов", столбцы текста программы нужно выровнять.
После такого "наведения порядка", при условии, что в тексте программы нет ошибок, ассемблирование пройдет успешно.
При загрузке текстов программ из других текстовых редакторов, нужно руководствоваться описанным выше принципом.
Если использовать ASM-редактор **Петра Высочанского** (в нем осуществляется автоматическое форматирование), то работу по выравниванию столбцов можно существенно упростить.

11. К вопросу о точном и стабильном формировании величин измерительных интервалов времени.

Недавно просмотрел один из форумов, на котором обсуждался этот вопрос и обнаружил ссылки на мой сайт, смысл которых в том, что я, якобы, утверждаю, что сформированный программными средствами интервал времени измерения будет нестабильным.
Да, он будет нестабильным, если программист поленился (или не знает, как это сделать) произвести "выравнивание" сценариев, которые исполняются в "внутри" интервала времени измерения.
Естественно, что такое может быть, но это свидетельствует только о недостаточно высоком уровне профессионализма.
В "Самоучителе...", я пытаюсь объяснить, как избежать этого "ляпа".
Можно абсолютно точно ("по нулям"), программно сформировать высокостабильные величины интервалов времени измерения.
Например, точно **100 000**, **1 000 000** или **10 000 000** машинных циклов
(или другое количество м.ц.), и без каких бы то ни было "гуляний" (если произведено "выравнивание").
Обращаю внимание → **машинных циклов**, а не микросекунд.
Количество машинных циклов, за время которых формируется интервал времени измерения, определяется только программой и не подвержено воздействию дестабилизирующих факторов (например, температуры).
При таком "раскладе", речь идет не о программной, а об аппаратной нестабильности.
Теоретически, при применении кварца на 4000000 Гц., 1 м.ц. = 1 мкс., но реально, формируется сигнал с частотой отличную от 4000000 Гц., плюс, дестабилизирующие факторы.
По этим причинам, требуются и коррекция частоты тактового генератора ПИКа (по эталонному частотомеру), и принятие мер по "сглаживанию" негативного влияния дестабилизирующих факторов (термокомпенсация или применение термостатирования).
Если принять меры по минимизации этих источников погрешностей/нестабильности, то с учетом сказанного ("по нулям" и "выравнивание"), можно "сваять" классное, измерительное устройство.

12. Для чего нужна команда andlw k ?

Чаще всего, эта команда (побитное "И" содержимого регистра **W** и 8-разрядной константы **k**) применяется при опросе клавиатуры, для "нейтрализации" тех битов байта, которые могут "исказить" результат опроса, ведь в подавляющем большинстве случаев, под клавиатуру задействуется всего-лишь несколько битов байта, а опрашивается-то весь байт.
В этих "вредоносных" битах, нужно просто выставить нули, не воздействуя при этом на "рабочие" биты.
Я это называю "нейтрализацией".
Давайте разбираться.
В подавляющем большинстве случаев, кнопки клавиатуры подключаются к выводам порта со

стороны младших его разрядов (**RA0,1...** или **RB0,1...**).
Это обусловлено тем, что после опроса состояний выводов порта (опроса клавиатуры), в большинстве случаев, отрабатывается процедура вычисляемого перехода, для которой редко когда требуется задействование битов с номерами 3...7.
Например, в ЧМ/ЦШ, клавиатура подключена к выводам **RA0**, **RA1**.
То есть, ПП опроса клавиатуры (включая и вычисляемый переход) рассчитана на работу с числами от **.00** до **.03**.
Но ведь эти 2 бита являются только частью байта, и нормальная работа ПП опроса клавиатуры будет обеспечена только при нулевых уровнях в остальных 6-ти битах.
В битах **№№ 5,6,7** нули будут присутствовать по определению (в **PIC16F84A**, биты **№№ 5,6,7** всегда читаются как нули).
А вот с битами **№№ 2,3,4** - сложнее: нужно разобраться с состояниями выводов **RA2, RA3, RA4**.
Проблем не будет, если эти выводы работают "на выход" и, перед опросом клавиатуры, все защелки порта А сброшены в **0**.
При этом, защелки выводов **RA0, RA1** тоже сбрасываются в **0**, но это "по барабану", так как по причине их работы "на вход", выходы этих защелок, от выводов **RA0**, **RA1** (к ним подключена клавиатура), отключены.
В этом случае все "ОК" и никаких проблем.
Именно такой случай и имеет место быть в программе ЧМ/ЦШ.
Усложняем задачу.
Задействуем порт В.
Подключаем 2-хкнопочную клавиатуру к выводам **RB0**, **RB1**.
К оставшимся шести выводам, подключаем выходы внешних устройств.
Таким образом, перед опросом клавиатуры, все выводы порта В должны быть настроены на работу "на вход".
Вывод: состояния выходов всех защелок, на момент опроса, перестают влиять на результат опроса состояния выводов порта В (выходы всех защелок отключены от соответствующих выводов порта В), и результат опроса будет определяться состоянием клавиатуры (**RB0, RB1**) и состояниями выходов внешних устройств (**RB2...7**).
Для нормального функционирования ПП вычисляемого перехода, с выводов **RB2...7**, необходимо считать нули.
Если это так, то все "ОК" и никаких проблем, но на практике, часто встречаются случаи, когда выходы одного или нескольких внешних устройств находятся в состоянии **1**.
В этом случае считается единица (единицы), после чего, в ПП вычисляемого перехода, рабочая точка программы будет "отфутболена" не туда, куда нужно, а "совсем в другую степь" (приращение **PC** будет больше ожидаемого), что есть не что иное, как нарушение работы программы.
Вопрос: "Как быть"?
Ответ: сразу же после опроса состояния выводов порта В (**movf PortB,W**) необходимо применить команду **andlw b'00000011'**.
Логика операции "И" такова, что если хотя бы один из битов равен **0**, то результат этой операции будет нулевым.
Посмотрите на константу.
Биты с **№№ 7...2** являются нулевыми, следовательно, биты с **№№ 7...2** результата логической операции "И" также будут нулевыми.
Причем, вне зависимости от того, какие уровни считались при опросе клавиатуры с выводов **RB2...7**. Что и требуется.
Это то, что я называю "нейтрализацией" (можно придумать и другое название).
Еще раз посмотрите на константу.
Биты с **№№ 0, 1** являются единицами, следовательно, биты с **№№ 0, 1** результата логической операции "И" будут в точности такими же, как и биты с **№№ 0, 1** результата опроса состояния выводов порта В.
Вывод: биты результата опроса выводов порта В, с **№№ 7...2**, "нейтрализованы".
То есть, после исполнения команды **andlw b'00000011'**, они гарантированно будут заменены нулями.
Состояния битов результата исполнения команды **andlw b'00000011'**, с **№№ 0, 1**, будут в точности повторять состояния битов с **№№ 0, 1** результата опроса порта В.
"Дешево и сердито".

Существуют и другие случаи применения команды **andlw k**.
Например, если производятся операции "в границах" полубайта, то ей можно "нейтрализовать" любой из полубайтов.
Также можно осуществить и выборочную "нейтрализацию" битов в байте.

13. О директиве INCLUDE и что такое goto $+N ?

Директива **INCLUDE** также, как и другие директивы, это всего-лишь "элемент удобства".
В "Самоучителе...", в учебно-тренировочных целях, "шапка" программы оформляется по принципу "прописки" только нужного, и не более того (ничего лишнего).
И в будущем я собираюсь действовать так же.
То есть, прежде чем переходить к "удобствам", нужно как следует "вжиться структуру шапки".
Для тех же, кто считает, что он в нее "вжился", и существует директива **INCLUDE**.
Воспользовавшись директивой **INCLUDE**, можно "прописать всё оптом" (кроме регистров общего назначения), но этот "опт окажется за кадром".
Естественно, что после этого, "масса шапки" существенно уменьшается.
А теперь детали.
При "вводе в эксплуатацию" директивы **INCLUDE**, "из недр" **MPLAB** извлекается и "подключается к работе" файл с расширением **.INC**, который есть не что иное, как "заготовка шапки" программы с уже "прописанными" в ней, по-максимуму, регистрами специального назначения (включая и "прописку" битов флагов, и "прописку" мест сохранения результатов операций).
Для каждого типа ПИКа имеется "свой" файл с расширением **.INC**, и все эти файлы находятся в папке **MPLAB**.
Откройте папку **MPLAB**, и Вы там их обнаружите целую "кучу".
Давайте просмотрим какой-нибудь из них, например, файл с названием **P16F84a.INC** (для **PIC16F84A**).
Откройте этот файл.
Если при его открытии, Вы увидите вопрос ***С помощью какой программы открыть?***, то выберете иконку программы **MPLAB** (а если ее нет в списке, то создайте).
После этого, в текстовом редакторе **MPLAB**, Вы увидите то, что после "ввода в эксплуатацию" директивы **INCLUDE**, "будет находиться за занавесом".
Для того чтобы воспользоваться этим "добром", нужно "соблюсти некоторые формальности":
1. В рабочей части программы, названия регистров и битов должны быть в точности такими же, как и в тексте файла с расширением **.INC**.
То есть, если написано **STATUS**, то и нужно писать **STATUS**, а не как-то иначе (например, **Status**), а иначе будете получать сообщения об ошибках.
2. Несколько изменилась форма представления битов конфигурации (бит с названием **...** включить/выключить), поэтому нет необходимости "ломать голову" над определением значения числа, выражающего их состояния (например, **03FF1H**).
Еще раз посмотрите в текст файла **P16F84a.INC** и обратите внимание на то, что регистры специального назначения 1-го банка (например, **OPTION_REG, TRIS...**) "прописаны" по своим фактическим адресам, то есть, **81h**, **8...h**, а не по адресам нулевого банка **01h**, **0...h**, которые я использую для написания текстов программ, не содержащих директивы **INCLUDE**.
То же самое относится и к файлам с расширением **.INC** других типов ПИКов.
Допустим, что текст программы, в которой используется директива **INCLUDE**, не содержит ошибок.
После этого "на гора будет выдан" список сообщений с итогом ***Build completed successfully*** (успешное ассемблирование).
Все эти сообщения являются информирующими (**Message**).
Они не являются ошибками (**Error**).
Типы, цифровые коды и содержание сообщений расписаны в **"MPASM. Руководство пользователя"**.
Появление таких информирующих сообщений обусловлено "пропиской", в файле с расширением **.INC**, фактических адресов регистров специального назначения 1-го банка.
Может возникнуть вопрос: "Почему в "Самоучителе...", во всех "шапках" программ, регистры специального назначения 1-го банка "прописываются" по адресам нулевого банка, а не по своим фактическим адресам?"
Ответ: чтобы избавиться от сообщений информативного характера типа **Message[302]**, которых достаточно много.

При этом, я руководствуюсь тем соображением, что на первых порах, не желательно распылять свое внимание на просмотр и анализ каких-то длинных списков сообщений.
При "вводе в эксплуатацию" директивы **INCLUDE**, должны быть применены специальные символы, и в "шапке" программы, написание названия INC-файла не должно отличаться от написания названия соответствующего INC-файла, "лежащего" в папке **MPLAB**.

Что такое goto $+N

Обращаю Ваше внимание на то, что речь идет не о символе **S**, а о символе **$** (доллар).
N – число "прыжка", которое может быть отображено в различных системах исчисления.
Например, исполнение команды **goto $+2** приведет к "прыжку" рабочей точки программы на 2-ю, после **goto $+2**, команду, по направлению сверху вниз.
Если вместо знака "**+**", использовать знак "**-**", то будет то же самое, но по направлению снизу вверх.
Подобрав числовое значение **N** и знак, можно "прыгнуть" (осуществить безусловный переход) туда, куда нужно.
При этом, отпадает необходимость в выставлении метки.
Команда **goto $** → "мертвяк" ("закольцовка" на саму себя, без возможности выхода).
Команда **goto $+1** заменяет два **NOP**а.

14. О выводе битов и байтов на выводы порта.

В случае вывода бита (битов), соответствующий (ие) вывод (ы) порта предварительно должен (ны) быть настроен (ы) на работу "на выход".
Если выводится весь байт, то все выводы порта предварительно должны быть настроены на работу "на выход".
"Настройка" направлений работы выводов порта осуществляется в регистре специального назначения **TRIS…**

Вывод бита с помощью команд BCF/BSF.

Для вывода бита, используются бит-ориентированные команды **BCF** (на выводе порта нулевой уровень) или **BSF** (на выводе порта единичный уровень).
"За один присест", можно вывести только один бит.
Если нужно последовательно изменить состояния, например, 3 битов, то необходимо так же последовательно, исполнить 3 бит-ориентированные команды **BCF**/**BSF**
Например, вывод нуля, на вывод **RB4** (извиняюсь за тавтологию), будет выглядеть так:
bcf PortB,4 (установить, на выходе защелки вывода **RB4**, нулевой уровень).
С учетом того, что **RB4** работает "на выход", это соответствует формированию, на выводе **RB4**, нулевого уровня.

Вывод байта с помощью команды MOVWF.

Для вывода байта используется байт-ориентированная команда **MOVWF**.
Предварительно, все выводы порта должны быть настроены на работу "на выход".
Сначала, байт (число) должен быть скопирован в регистр **W** из какого-нибудь регистра общего назначения или этот байт (число) должен быть константой, записанной в регистр **W** с помощью команды **MOVLW**.
Например: **movwf PortB**.
"Расшифровка": установка на выходах всех защелок порта B уровней, предварительно записанных в регистр **W** (при работе "на выход", выходы защелок подключены к выводам порта).
Еще проще: вывод содержимого регистра **W** в порт B.
Если в регистр **W** была записана константа, то она и будет выведена в порт B.
То же самое можно сказать и о случае копирования, в регистр **W**, содержимого какого-то регистра общего назначения, например, с названием **ABCD** (**movf ABCD,W**).
В большинстве случаев, в регистре **ABCD** "лежит" результат "долгой и кропотливой работы" программы (в динамике).

"Быстрый" вывод группы битов с количеством бОльшим 1, но меньшим 8-ми.

Особое место занимает случай, когда "за один присест" (без использования бит-ориентированных команд), в порт нужно вывести количество битов более одного, но менее восьми.
Предположим:
На выводы порта B с номерами **0,1,2,3** нужно вывести полубайт.
К выводам порта B с номерами **4,5,6,7** подключены выходы внешних устройств.

Что нужно сделать?
Перед выводом полубайта, необходимо "настроить" биты регистра **TrisB** следующим образом: выставить в битах с **№№0,1,2,3** нули (работа выводов **RB0,1,2,3** "на выход"), а в битах с **№№4,5,6,7**, выставить единицы (работа выводов **RB4,5,6,7** "на вход").
Это выглядит так:

```
bsf     Status,RP0  ; выбор 1-го банка (или bsf Status,5)
movlw   b'11110000' ; запись константы в аккумулятор
movwf   TrisB       ; копирование числа 11110000, из W,
                    ; в регистр TrisB
```

Если в дальнейшем не требуется производить каких-то операций с содержимым регистра (регистров) специального назначения 1-го банка, то после команды **movwf TrisB**, нужно выбрать 0-й банк (**bcf Status,RP0** или **bcf Status,5**).
Всё. Выводы порта B "настроены" так, как было указано выше, и теперь можно "выводить наружу" полубайт.
Вопрос: "С выводами, настроенными на работу **на выход**, все ясно. На них будет выводиться полубайт. А не повлияет ли процесс вывода полубайта на состояния остальных 4-х выводов, настроенных на работу **на вход**?"
Ответ: нет, не повлияет (хотя на выходах защелок выводов **RB4,5,6,7** выставятся какие-то уровни. В данном случае, не важно какие) по той причине, что выходы этих защелок электрически отключены от выводов **RB4,5,6,7** (работа "на вход").
Уровни, присутствующие на выводах **RB4,5,6,7**, если в дальнейшем они не будут перестраиваться на работу "на выход", полностью определяются состояниями выходов внешних устройств, подключенных к ним.
В регистр **W** должна быть записана либо константа (**movlw k**, где **k** - константа), либо в него должно быть скопировано содержимое регистра общего назначения, в котором хранится результат предшествующей работы программы (**movf ABCD,W**).
Затем, **movwf PortB**, и дело сделано.
Вопрос: "А если нужно, например, вывести три бита. И не соседних друг с другом, а допустим, с **№№1,4,7**?"
Ответ: они выводятся так же, как и в описанном выше случае, только нужно внести коррективы в значение константы, записываемой в регистр **TrisB**.
Для случая работы остальных выводов порта B "на вход", она будет выглядеть так: **01101101**
Существуют и другие разновидности вывода "наружу" группы битов с количеством бОльшим 1, но меньшим 8-ми.
Руководствуясь сказанным выше, плюс немного смекалки, можно реализовать любую разновидность вывода.

15. Пример организации 3-байтного счетчика.

```
                ..................  ; Команды программы.
                movlw       L       ; Запись константы L
                movwf       SecL    ; в регистр SecL.
                movlw       M       ; Запись константы M
                movwf       SecM    ; в регистр SecM.
                movlw       H       ; Запись константы H
                movwf       SecH    ; в регистр SecH.
                ..................  ; Другие команды программы
                ..................  ; (могут быть, а могут и не быть).
PAUSE           decfsz      SecL,F  ;
                goto        PAUSE   ; Классический
                decfsz      SecM,F  ; 3-х байтный
                goto        PAUSE   ; вычитающий
                decfsz      SecH,F  ; счетчик.
                goto        PAUSE   ;
                goto ..............  ; Переход туда, куда нужно или без этой команды,
                ..................  ; если работа происходит на "линейном" участке
                ..................  ; программы.
```

При помощи такого счетчика можно обеспечить достаточно длительные задержки.

По такому же принципу можно "сконструировать" счетчик любой "байтности".
То, что Вы видите, это классический 3-байтный вычитающий счетчик.
Заменив **decfsz** на **incfsz**, получите суммирующий счетчик.
Применив и **decfsz**, и **incfsz**, получите комбинированный счетчик.
Номиналы констант зависят и от конструкции счетчика (в том числе и с учетом "врезок"), и от величины требуемой задержки.