

ВВЕДЕНИЕ

Глава 1. ТЕОРИЯ, ПОНЯТИЯ, ОПРЕДЕЛЕНИЯ

Память

Регистры

Системы счисления

Адрес регистра

Банки памяти

Регистры специального назначения

Регистры общего назначения

Как работает микроконтроллер?

Что такое сигнал?

Формат записи чисел

Глава 2. КОМАНДЫ АССЕМБЛЕРА

Сопоставление "имен" и "чисел"

Команды сложения и вычитания регистров

Команды определения бита

Команды взаимодействия с аккумулятором W

Команды сложения и вычитания констант

Команды очистки регистров F и W (обнуления)

Пустышки и метки

Команды переходов

Вопросы для самопроверки

Команды счётчики

Переход по событиям в счётчиках

Переход по результатам бит-проверки

Флаги как индикаторы событий

Команды сравнения

Команды сдвига битов в регистре

Глава 3. СОСТАВЛЕНИЕ ТЕКСТА ПРОГРАММЫ НА АССЕМБЛЕРЕ В MPLAB

Установка и подготовка к работе MPLAB

Создание проекта и подключение файла с программой

Структура текста программы

Правила оформления программы

Конфигурирование МК. Директива __CONFIG

Особенности сопоставления имен и чисел. Прямая и косвенная адресация

Циклическая концепция программы

Модульная структура программы

Понятие о времени исполнения программы

Задержки в программе и их расчёт

Компиляция и устранение ошибок

Глава 4. МАКЕТНАЯ ПЛАТА. ПРОГРАММАТОР

Обозначение ножек микроконтроллера

Функциональное назначение ножек

Документация на микроконтроллер PIC16F84A

Макетная плата и её назначение

Схема JDM-совместимого NTV-программатора

Инструкция по прошивке микроконтроллера

Глава 5. ЭЛЕМЕНТАРНЫЕ БАЗОВЫЕ ПРОЕКТЫ

Теория и практика работы портов МК

Пример 1. Мигающие светодиоды

Пример 2. "Бегущий огонь" и "бегущая тень"

Пример 3. Включение символов на индикаторе

Пример 4. Отслеживание нажатия кнопки

Пример 5. Кнопка в режиме переключателя. Антидребезг

Пример 6. Работа нескольких кнопок. Многозадачность

Пример 7. Уменьшение и увеличение значений кнопками

Пример 8. Энкодер и шаттл: ввод цифровой информации

Пример 9. Работа с энергонезависимой памятью (ПЗУ)

Глава 6. ИНСТРУМЕНТЫ MPLAB

Симулятор MPLAB SIM

Анализ регистров общего и специального назначения

Измерение времени исполнения программы

Глава 7. СОПРЯЖЕНИЕ МИКРОКОНТРОЛЛЕРА С ВНЕШНИМИ УСТРОЙСТВАМИ

Описание интерфейса RS-232

Передача данных в сторону компьютера

О кодовой таблице ANSI

Электрическое сопряжение с ПК

Работа с терминалом на ПК

Приём данных от ПК на стороне МК

ЭТО не КОНЕЦ

Тимофей Носов

ВВЕДЕНИЕ

Тема микроконтроллеров, их программирования, конструкции с их использованием очень долго для нас оставались чем-то фантастичным и непостижимым. До сих пор нет доступной литературы, рассчитанной на новичков, самостоятельно начинающих почти с нуля.

Посмотрев с этой точки зрения на появившиеся в последние несколько лет посвященные микроконтроллерам книги, а также на публикации в журналах, становится понятным, что практически все они направлены на тех, кто уже ориентируется в этой предметной области. Статей и книг, рассчитанных на новичков, и позволяющих им шаг за шагом освоить микроконтроллеры, не перегружая их головы раньше времени важными, но необязательными в первый момент подробностями, увы, нет. Так родилась идея написать самоучитель для начинающих, знакомство с которым позволило бы им осознать, что такое микроконтроллеры, как они устроены, как функционируют, как писать, отлаживать и заносить в них программы, Как подключать к ним другие устройства и т.д.

Всех, кто попытается по нашему самоучителю пройти путь обучения, мы просим не стесняться задавать вопросы, если что то окажется непонятным или плохо изложенным – наш электронный адрес ntv1978@mail.ru .

Любой, даже самый сложный вопрос, можно объяснить просто и доступно. Постараемся это сделать и мы. В качестве объекта изучения мы с вами выберем микроконтроллеры семейства PIC. Предполагаем, что некоторые из тех, кто прочел эти строки, уже поморщились – зачем, дескать, писать об этом, сейчас есть много гораздо более интересных контроллеров. Да, есть. Но, во-первых, нам легче объяснять материал на основе того, что мы знаем очень хорошо. Во-вторых, все, кто разобрался хотя бы с одним контроллером, после этого всегда в состоянии самостоятельно разобраться с любым иным – было бы время и желание (или необходимость). В-третьих, с этими контроллерами по-прежнему работает не меньше разработчиков, чем с AVR-контроллерами, не говоря уже о любых других, а Atmel и Analog Devices в последнее время предоставили в наше с вами распоряжение еще более совершенные образцы контроллеров этого семейства. А в-четвертых, и это главное, те, кто поморщился – это люди, уже разбирающиеся в микроконтроллерах.

Господа разбирающиеся! Этот самоучитель не для вас! Здесь вы вряд ли найдете что-то новое для себя. Читать этот материал мы вам рекомендую лишь в том случае, если вы замыслите написать подобные статьи по другим типам контроллеров. Тогда читайте то, что написано нами, фиксируйте, что изложено удачно, а что нет, и готовьте свои статьи с учетом возможных огрехов, увы, неизбежных в первой редакции такого материала. Ну а мы с теми, кто еще не разбирается в микроконтроллерах, потихоньку двинемся дальше.

Глава 1. ТЕОРИЯ, ПОНЯТИЯ, ОПРЕДЕЛЕНИЯ

В этой главе мы осветим несколько простых и одновременно сложных для понимания сущностей. Речь пойдет о памяти, регистрах, системах счисления, адресе регистра, о банках памяти, о регистрах специального и общего назначения. Мы ответим на вопросы "Как работает микроконтроллер?" и "Что такое сигнал?". Отдельно рассмотрим форматы записи чисел.

С одной стороны, оглядываясь по прошествию многих лет на эти знания, мне они кажутся не существенными. А с другой стороны, когда я начинал изучать микроконтроллеры, без отсутствия такого представления мне было бы сложно дальше продолжать освоение МК. Желаю вам успешно освоить эти сущности.

Память

В каждом микроконтроллере (далее МК) есть области памяти. Память бывает трёх типов: память программы, оперативная память, энергонезависимая память.

В память программы загружаются строчки текста программы, которые мы далее составим и научимся загружать. Загруженный текст программы не изменяется и не пропадает после выключения питания МК. Размер памяти программ определяется типом используемого МК. Для PIC16F84A размер памяти программ составляет 1024 строчки.

Оперативная память используется для обращения к ней из текста программы. При включении МК ячейки оперативной памяти пусты (точнее, их содержимое неизвестно). Данные (числа) загружаются в оперативную память в ходе выполнения программы. Вообще, оперативная память МК на практике используется для временного хранения данных. Размер оперативной памяти относительно памяти программ гораздо меньше. В PIC16F84A оперативная память состоит из 36 ячеек.

Энергонезависимая память содержит данные, которые при выключении питания МК не пропадают. Содержимое энергонезависимой памяти мы можем определить в процессе написания программы, а также содержимое может измениться в ходе выполнения программы в МК. В PIC16F84A энергонезависимая память состоит из 64 ячеек.

Регистры

Память состоит из так называемых ячеек или иначе РЕГИСТРОВ, соответственно числа записываются в регистры. Далее, говоря о памяти, мы будем подразумевать оперативную и энергонезависимую память. Память программ организована иначе; пока мы не будем её рассматривать, т.к. знания о ней менее актуальны.

Для простоты восприятия память лучше представить в виде таблицы, которая состоит из 16 столбцов с некоторым количеством строк и у которой в каждую ячейку можно записать число.

19	28	37	46	55	64	73	82	91	0	11	22	33	44	55	66
77	88	99	0	201	202	203	204	205	206	207	208	209	0	1	2
3	4	5	6	7	8	9	0	0	0	0	0	0	0	0	0

Системы счисления

В регистр может быть записано одно положительное число от 0 до 255. Машинная математика, в т.ч. и математика МК, не может оперировать десятичными числами, для этого используется так называемая двоичная система счисления (или иначе – бинарная).

Десятичное число 255 в двоичной системе выглядит как "11111111", т.е. восемь единиц, а двоичное число "0" – как восемь нулей "00000000". Следует заметить, что обозначение нуля одним символом "0", в любой системе счисления равно нулю. Таким образом, "регистр" необходимо рассмотреть детальнее, т.к. в него записывается не десятичное число, а двоичное.

Запомним: "мы говорим о десятичных числах в МК, а подразумеваем двоичные". Ниже рисунок памяти в реальности

00010011	00011100	00100101	00101110	00110111	
01001101	01011000	01100011	00000000	11001001	

Такие двоичные числа называют восьмибитными числами или байт.

Запомните: 1 БАЙТ = 8 БИТ.

Таким образом, 1 байт представляет собой последовательность нулей и единиц (или набор битов), например, 11011000, где количество нулей и единиц равно восьми, отсюда и название – "восьмибитное".

Нумерация битов идёт справа налево от нуля до семи. В нашем примере нулевой бит равен нулю, а седьмой бит равен единице. Это так называемая бинарная система счисления, где используются две цифры "0" и "1". Перебирая нули и единицы в байте, мы можем получить 255 комбинаций, т.е. любое число на интервале от 0 до 255. Таким образом, в регистр мы можем записать десятичное число от 0 до 255.

Необходимо понять, что в регистр физически записывается число в восьмибитном представлении, работу регистра проще представить как набор из 8-ми крошечных выключателей, каждый из которых может быть включен или выключен.

Адрес регистра

Каждый регистр имеет свой порядковый номер, т.н. адрес. АДРЕС РЕГИСТРА традиционно обозначается числом из шестнадцатеричной системы счисления, например, 1A. Шестнадцатеричное число представляет собой комбинацию 16 символов: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. В нашем случае 1A – это десятичное число 26. Перевод чисел из одной системы в другую несложно сделать в стандартном калькуляторе Windows (в настройках калькулятора выбрать вид "инженерный"). Рекомендую скачать [конвертер BinHexDec](#) – многое в системах счисления станет понятнее.

Исходя из шестнадцатеричной системы счисления, для лучшего восприятия, память нарисована в виде 16-ти столбцов. Нумерация ячеек памяти изображена в таблице ниже.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F

Наложив предыдущие таблицы друг на друга, мы можем обнаружить, что регистр с адресом 0 содержит число 19; а регистр с адресом 12 – содержит число 99.

Банки памяти

Регистры в МК объединены в группы, так называемые **БАНКИ ПАМЯТИ**. Банки нумеруются десятичными числами и называются BANK0, BANK1 и т.д. Количество банков зависит от количества регистров, т.е. зависит от размера памяти и её организации в МК. Например, BANK0 содержит регистры с адресами в диапазоне от 0 до 127, BANK1 содержит регистры с адресами в диапазоне от 128 до 255. Для наглядности здесь диапазоны указаны в десятичном представлении.

В МК различают регистры специального и общего назначения.

Регистры специального назначения

В РЕГИСТРАХ СПЕЦИАЛЬНОГО НАЗНАЧЕНИЯ содержится служебная информация, определяющая настройки работы МК. Под настройками понимается запись восьмибитных чисел в регистры специального назначения, где каждый бит в этих числах определяет ту или иную настройку.

Регистры специального назначения находятся в области оперативной памяти.

Регистры специального назначения имеют жестко определенные адреса и регламентированные наименования, которые определены в документации для того или иного МК. Как правило, в разных МК адреса регистров специального назначения совпадают, что подтверждает простоту переноса программ из одних МК в другие. Регистры специального назначения в оперативной памяти МК как правило расположены в самом начале, например, имеется 10 регистров специального назначения, занимающих диапазон адресов от 00 до 09.

Регистры общего назначения

РЕГИСТРЫ ОБЩЕГО НАЗНАЧЕНИЯ используются для записи и хранения переменных и констант. Регистры общего назначения также находятся в области оперативной памяти и нумерация их адресов продолжает нумерацию регистров специального назначения.

Ранее мы определили, что каждый регистр имеет адрес, но мы не определили, что такое имена регистров или наименования. Для удобства программирования "числовым" адресам регистров сопоставляют "буквенные" имена. Например, регистру с адресом 03 можно дать имя STATUS (о том, как сделать сопоставление см. в главе 2).

Необходимо запомнить:

- регистры специального назначения имеют определенные имена, которые отражены в документации; не рекомендуется регистрам специального назначения присваивать другие имена (например, регистр **STATUS** переименовать в **STAT**);
- регистрам общего назначения, как правило, не присваивают имена, определенные для регистров специального назначения;
- имена регистров не должны совпадать по написанию с командами;
- в тексте программы необходимо придерживаться единообразного написания имени. Например, имя **PER**, **Per** и **per** разные имена; рекомендую придерживаться единого стиля в написании;
- имена набираются на английской раскладке без пробелов "разумной" длины, допускается символ нижнего подчеркивания "_" и цифры.

Как работает микроконтроллер?

Работа МК заключается в сравнении и изменении чисел в регистрах, в сравнении и изменении отдельных битов в регистрах. Сравнение и изменение происходит в соответствии с программой. Программа – это последовательность строк с командами. Команды обращаются к тем или иным регистрам и модифицируют их или изменяют их содержимое, точнее, изменяют числа; еще точнее – изменяют биты восьмибитных чисел.

А теперь **самое главное** о работе МК – изменение определенных битов в определенных регистрах приводит к тому, что на определенных ножках (выводах) МК появляются сигналы. **Наша задача** – получить эти сигналы в определенный момент времени и в определенной последовательности для того, чтобы МК выполнял полезные функции.

МК может не только выдавать сигналы на ножки, т.е. "дергать" ножками, но и **"реагировать" на внешние сигналы**. Необходимо отметить, что данные в некоторых регистрах специального назначения могут изменяться не только в результате работы программы, но и в результате внешнего воздействия, например, по нажатию кнопки или в результате определенных событий в МК (например, снижения напряжения).

Что такое сигнал?

Упрощенно говоря, под сигналом понимается одно из двух состояний на ножке. **Нулевой сигнал или сигнал Низкого Логического Уровня (НЛУ)** близок к нулю 0 вольт относительно минуса питания. **Единичный сигнал или сигнал высокого логического уровня (ВЛУ)** соответствует напряжению питания или +5 вольт относительно минуса питания.

Напряжение на ножке выше чем 60% напряжения питания МК гарантировано воспринимается им как ВЛУ

или "1" – высокий логический уровень. Напряжение на ножке ниже чем 20% напряжения питания МК гарантировано воспринимается им как НЛУ или "0" – низкий логический уровень. Эти пороговые уровни для напряжения питания 5 вольт будут 3 и 1 вольт.

Формат записи чисел

Ранее мы упоминали о десятичной, двоичной и шестнадцатеричной системе счисления. Например, набор символов "100" можно отнести к любой из этих систем:

100 – в нашем обычном восприятии;

4 – в двоичной системе счисления;

256 – в шестнадцатеричной.

В связи с этим, требуется иной способ записи, который однозначно определял бы формат числа. Для этого, в общих случаях, символы числа ставят в одинарные кавычки, а перед ним ставят одну из английских букв: В (или b) – для бинарного числа, Н (или h) – для шестнадцатеричного, D (или d) – для десятичного (вспомните [конвертер BinHexDec](#)).

См. таблицу ниже – всё станет понятно.

Формат	Синтаксис	Пример записи
10 десятичный	Dчисло .число	D202 или d202 .202
2 двоичный	Вчисло	B00000101 или B101 или b101
16 шестнадцатеричный	Нчисло 0хчисло	H03A или H3A или h3A 0x03A или 0x3A

Как видно, есть и альтернативные варианты: запись с точкой и запись вида 0х... Первые незначащие нули откидываются.

Какой формат записи использовать, зависит от ситуации и восприятия. Все эти примеры записи используются при написании текста программ в MPLAB (об этом позже). На бытовом уровне понятнее использовать десятичный (для записи мы используем точку с числом), но есть случаи, когда нужен другой формат. Например, адреса регистров традиционно пишут только в шестнадцатеричной форме. При работе с битами регистров нагляднее использовать двоичную запись.

Необходимо отметить, что от нуля до девяти включительно 16-ричная и 10-тичная системы исчисления одинаковы в написании. Если записываем число без атрибутов (В,Н,D и "точка"), то по умолчанию оно будет считаться 16-ричным, но, как мы отметили, от 0 до 9 в десятичном виде оно точно такое же. После числа девять появляются расхождения.

И напоследок. Попрактикуйтесь с переводом чисел из одной системы в другую; необходимо уяснить суть регистра – его содержимое, адрес и имя.

Глава 2. КОМАНДЫ АССЕМБЛЕРА

Ассемблер – язык программирования из **35 команд**.

Здесь в мизерных дозах мы научимся читать некоторые команды, понимать физиологию их выполнения, рассмотрим примеры и постараемся использовать простые термины. Для облегчения изучения команд и для облегчения перехода к практическому программированию команды будут рассмотрены в следующем порядке:

- директива сопоставления имен и чисел;
- команды сложения и вычитания регистров;

- команды определения бита;
- команды взаимодействия с аккумулятором W;
- команды сложения и вычитания констант;
- команды очистки регистров F и W (обнуления);
- пустышки и метки;
- команды переходов;
- команды счётчики;
- переходы по событиям в счётчиках;
- переходы по результатам бит-проверки;
- команды сравнения.

Группировка команд по таким критериям расширяет традиционную группировку, улучшает понимание команд и их предназначение. Необходимо понять не только смысл команд, но связь между командами, последовательность их выполнения, причинные и следственные изменения хода выполнения команд.

Перед началом изучения этой главы скачайте из раздела [Документация](#) русский даташит на PIC16F628. Поместите ярлык на рабочий стол. Распечатайте в виде брошюрки-шпоргалки стр. 105-124 и параллельно изучайте.

Сопоставление "имен" и "чисел"

Начнем с самого простого – с директивы EQU, которая машинным числам определяет человеческие имена. Директивы – это команды, которые устанавливают определенный порядок работы программы.

Глядя на машинную математику МК, становится очень тоскливо от обилия чисел в разных системах счисления: значительное количество адресов регистров помноженное на восемь бит в каждом регистре. Но это не должно пугать, т.к. на практике всё гораздо проще. Регистрам и битам в регистрах, с которыми мы будем работать, присваиваются имена.

Теперь нам пора поработать с реальными фрагментами программы; все фрагменты, которые будут использоваться в примерах – взаимосвязаны между собой. Поэтому, если какой-то фрагмент непонятен, нужно читать предыдущие примеры.

Пример сопоставления имен:

;сопоставление значений селектора (о селекторе чуть позже)

W	EQU	0x0
F	EQU	0x1

;сопоставление адресов регистров

INDF	EQU	H0000
TMR0	EQU	H0001
PORTB	EQU	H0006
KLON	EQU	H000C
KLOP	EQU	H000D

;сопоставление номеров битов в регистре

Z	EQU	2
DC	EQU	1
C	EQU	0

В примере специально был использован разный формат записи чисел для того, чтобы вы привыкали к различным вариантам записи.

Внимание: адреса регистров рекомендуем указывать только в 16-ричном формате.

Итак, для простоты понимания **сопоставление** – это назначение числам каких либо имен (наборов символов). Проще понимать это иначе: мы назначаем именам какие-то числа, которые, по нашему мнению будут использоваться в программе ("дом Ивановых – это дом номер 21А, а дом Петровых – это дом 19Б; мы переселяемся из дома Петровых в дом Ивановых, а по машинному – переселяемся из 19Б в 21А").

В последующем вы поймете, что сопоставление по сути не важный момент в программировании. Вы можете какие-то регистры и биты сопоставить, какие-то не сопоставить. Так или иначе, в тексте программы в любой момент времени вам ни кто не препятствует обратиться к регистрам или битам либо по их именам, либо по их порядковым адресам или номерам, определенных в документации МК.

Команды сложения и вычитания регистров

Итак, рассмотрим конструкцию **XXXXX F,D** , где

XXXXX – слово команды + пробел;

F – имя или адрес какого-либо регистра, с которым будет работать команда + символ запятой;

D – так называемый селектор, который может быть равен 0 или 1. Если **D=1** то результат сохраняется в регистре **F**. Если **D=0** , то результат сохраняется в **регистре W**. Внимание, если **D** не указано, то ассемблер по умолчанию предполагает, что **D=1**. Напоминаю – регистр **F** это ячейка оперативной памяти.

Регистр W – это так называемый рабочий регистр, который не имеет адреса; регистр **W** проще представить как "виртуальный" регистр, который в командах выступает посредником, через него или с помощью него выполняют те или иные операции с "реальными" регистрами. Иначе **W** называют аккумулятором.

Например, необходимо сложить два регистра **F1** и **F2** (**F1** и **F2** в этом примере имена регистров; могут быть другие имена, например, **KLON** и **KLOP**). Для этого содержимое **F1** надо скопировать в аккумулятор **W**, затем сложить содержимое аккумулятора **W** с регистром **F2** (работа с аккумулятором **W** рассмотрена ниже).

Команда СЛОЖЕНИЯ (F + W) в общем виде выглядит как ADDWF F,D
--

Команда ВЫЧИТАНИЯ (F – W) в общем виде выглядит как SUBWF F,D

Внимание: вычитание строго из **F – W**; других вариантов нет!

Для удобства значениям селектора сопоставляются имена, а именно "**W это 0**" и "**F это 1**" , это хорошо видно в примере сопоставления выше. Таким образом, команда **ADDWF F2,1** может иметь вид **ADDWF F2,F** , а команда **ADDWF F2,0** будет иметь вид **ADDWF F2,W** . Это означает, что если в селекторе указан символ **W** , то результат команды записывается в аккумулятор **W**. Если в селекторе указан символ **F** – то результат команды записывается в регистр, который был указан в команде (перечитайте этот абзац ещё раз).

Продолжим пример, рассмотренный выше. Нам надо сложить содержимое регистра **KLON** с содержимым **аккумулятора W** и результат операции сложения поместить в регистр **KLON** (рекомендую команды именно таким образом проговаривать).

ADDWF KLON,F ; запись с сопоставленными именами

ADDWF H000C,0x1 ; тоже самое, но в числах

ADDWF H000C ; тоже самое, но с учетом умолчания

ADDWF H000C,DC ; тоже самое, но с приколом

А причем тут символы "DC" в строке? Ответ прост: символам "DC" в предыдущем примере мы сопоставили число 1. Здесь ошибки нет, но лучше так не прикалываться, чтобы самому не запутаться. Какую форму записи использовать – выбирать вам. Если в тексте программы регистр используется пару раз, можно его не сопоставлять с именем (не называть), а просто в строке команды указать его адрес.

Примеры с вычитанием. Надо из содержимого регистра **KLOP** вычесть содержимое **аккумулятора W** и результат операции вычитания поместить в **аккумулятор W**.

SUBWF KLOP,W ; запись с использованием сопоставленных имен

SUBWF H000D,0 ; тоже самое, но в числах

SUBWF H000D,W ; тоже самое, но с использованием числа и имени

Таким образом, строки программы представляет из себя команды, которые фактически обращаются к числам через имена.

Сопоставление номеров битов в регистре – также оправданная необходимость. Например, нам надо управлять 8-ю ножками, устанавливая на них +5В или 0В (напоминаю: +5В – это логическая единица; 0В – это логический ноль). Эти ножки – реальные выводы МК, они имеют нумерацию, например, с 1-го по 8-й вывод (или другую, сейчас это не важно). А вот в регистре, который управляет восемью ножками, биты пронумерованы от 0 до 7. Например, 0-й бит управляет 1-й ножкой, 1-й бит – 2-й ножкой и т.д. до 7-й бит – 8-й ножкой. Всё это заставляет сосредотачивать внимание и порой тратить время на перепроверку правильности записи.

Проще так – сопоставить реальные выводы МК с виртуальными битами.

NOGA1 EQU H0000

NOGA2 EQU H0001

...

NOGA8 EQU H0007

Просто, понятно, и даже немного вульгарно.

Напоминаю, что регистры нумеруются от нуля до семи, справа налево.

NOGA8	NOGA7	NOGA6	NOGA5	NOGA4	NOGA3	NOGA2	NOGA1
бит N7	бит N6	бит N5	бит N4	бит N3	бит N2	бит N1	бит N0
0	0	0	1	1	1	1	1

Например, запись числа **00011111** в регистр с названием **PORTB** означает, что на восьми ножках микросхемы, относящихся к "Порту Б" с 1-й по 5-ю ножку, будет установлен высокий уровень сигнала, а на остальных 6,7 и 8-й ножке будет установлен низкий уровень сигнала.

Команды определения бита

Рассмотрим конструкцию **XXXXX F,B** где

XXXXX – слово команды + пробел;

F – имя или адрес какого-либо регистра, с которым будет работать команда;

B – номер бита в регистре, т.е. число от нуля до семи.

Мы знаем, что бит может принимать одно из двух значений, т.е. быть либо равен единице, либо равен нулю. Для упрощения говорят так: "установить бит", "устанавливается бит", "установлен бит", где под установкой понимают перевод бита в состояние "единица"; противоположная терминология – "опустить бит", "обнулить бит", "установить бит в ноль", "сбросить бит", т.е. перевести бит в состояние "ноль".

Команда УСТАНОВКИ БИТА в общем виде выглядит как BSF F,B
--

Команда ОБНУЛЕНИЯ БИТА в общем виде выглядит как BCF F,B
--

Команда **BSF** устанавливает бит в единицу.

Команда **BCF** опускает бит в ноль.

Прежде чем продолжить рассмотрение практического примера записи строчек программы, вспомните о регистрах специального назначения; если не вспомнили, найдите в предыдущем разделе и перечитайте. Если прочитали, то начнем управление первой ножкой МК и установим на ней высокий уровень сигнала. Нам надо установить единицу в нулевом бите в регистре **PORTB**.

```
BSF PORTB,NOGA1      ; запись с сопоставленными именами
BSF H0006,H0000       ; запись в числах
BSF H0006,0           ; еще вариант записи в другом формате
BSF PORTB,0           ; еще вариант записи в другом формате
```

А теперь несколько вариантов по установлению сигнала низкого уровня на восьмой ножке МК. Нам надо установить ноль в седьмом бите в регистре **PORTB**.

```
BCF PORTB,NOGA8      ; запись с сопоставленными именами
BCF H0006,H0007       ; запись в числах
BCF H0006,7           ; еще вариант записи в другом формате
BCF PORTB,7           ; еще вариант записи в другом формате
```

Надеюсь, что вы почувствовали разницу в написании.

Команды взаимодействия с аккумулятором W

Рассматривая БАЙТориентированные команды сложения и вычитания, мы говорили о регистре-посреднике – регистре-аккумуляторе **W**. Закономерен вопрос: каким образом записать число в регистр аккумулятор W ?

У нас есть три варианта:

- 1) в регистр **W** можно копировать число из другого регистра;
- 2) в регистр **W** можно записать число в диапазоне от 0 до 255;
- 3) в регистр **W** число может записаться в результате операции сложения или вычитания. Надеюсь, что байт-ориентированные команды усвоены.

Мы будем рассматривать 1й и 2й варианты. Строго говоря 2й вариант относится к операциям с константами. Но мы решили осветить его именно в этом месте, т.к. проще будет понять отличия. Дополнительно будет изучен вариант копирования числа из аккумулятора **W** в регистр.

Копировать в РЕГИСТР W число из регистра F MOVF F,W
--

Копировать в РЕГИСТР W обычное число MOVLW K
--

Копировать число из РЕГИСТРА W в регистр F MOVWF F

MOVF KLOP,W ; копировать в регистр **W** число из регистра **KLOP**

MOVLW .255 ; копировать в регистр **W** число 255

MOVWF KNOP ; копировать число из регистра **W** в регистр **KNOP**

Внимательно посмотрите на эти команды и почувствуйте разницу, т.е. поймите, какой командой можно скопировать число из регистра, а какой командой можно скопировать в **W** число – это разные вещи.

Команда копирования числа из регистра **W** в другой регистр не требует комментариев.

Есть случай записи команды "**MOVF KLOP,F**", при этом результат помещается назад в регистр **KLOP**. Казалось бы – глупость, но это не так. Этой командой проверяют значение регистра на равенство нулю. Как это делается – будет рассмотрено в частных случаях.

Команды сложения и вычитания констант

Рассмотрим конструкцию **XXXXX K** где

XXXXX – слово команды + пробел;

K – константа-число в диапазоне от 0 до 255.

Команда **СЛОЖЕНИЯ (K + W)** в общем виде выглядит как **ADDLW K**

Команда **ВЫЧИТАНИЯ (K – W)** в общем виде выглядит как **SUBLW K**

Внимание: результат операций **ADDLW K** и **SUBLW K** помещается в аккумулятор W. Вычитание строго **K – W**, других вариантов нет!

ADDLW .123 ; к 123 прибавить число из аккумулятора

SUBLW .255 ; из 255 вычесть число из аккумулятора

Команды очистки регистров F и W (обнуления)

Команда **ОЧИСТКИ РЕГИСТРА F** в общем виде выглядит как **CLRF F**

Команда **ОЧИСТКИ АККУМУЛЯТОРА W** выглядит как **CLRW**

CLRF CLOP ; очистить регистр CLOP или
; установить в 0 все биты регистра KLOP

CLRW ; очистить аккумулятор W
; установить в 0 все биты регистра W

Пустышки и метки

ПУСТЫШКА – команда, которая ничего не делает, но на её выполнение уходит время, пустышка – убийца времени. Обозначается как **NOP**.

NOP ; команда пустышка

NOP ; команда пустышка

... вот так и убивается время. Казалось бы, команда паразит, но она нужна в отдельных случаях для корректировки времени выполнения программы.

МЕТКА – это слово, которая ставится перед командой. Метка используется командами переходов для перемещения от одной команды к другой, как вперед по тексту программы, так и назад. Метка набирается в английской раскладке без пробелов "разумной" длины, допускается символ нижнего подчеркивания "_" и цифры. Имена меток не могут повторяться.

Metka1	CLRW	; очистить аккумулятор W
	NOP	; команда пустышка
Metka2	ADDLW .123	; к 123 прибавить число из аккумулятора
	NOP	; команда пустышка

Т.к. строчки программы в МК выполняются последовательно, одна за другой, метки являются важными элементами в тексте программы. Без них сложно организовать логику работы программы. Мы говорим сложно, т.к. вместо переходов по меткам можно принудительно изменять значение аппаратного счетчика команд, который находится в регистре специального назначения PSL (адрес hA).

Команды переходов

Команды переходов используются для перемещения по тексту программы от одной строчки к другой. Для переходов используются метки.

Ни одна программа не может существовать без переходов, т.к. переходы обеспечивают непрерывность и цикличность программы (подробнее об этом в следующем разделе).

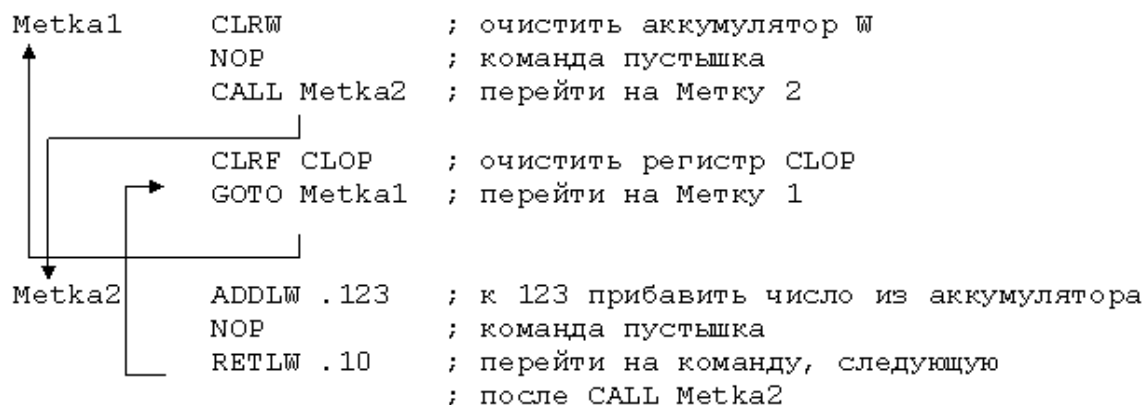
Различаются команды переходов без возврата и с возвратом.

ПЕРЕХОД НА МЕТКУ в общем виде выглядит как **GOTO МЕТКА**

СПЕЦПЕРЕХОД НА МЕТКУ в общем виде выглядит как **CALL МЕТКА**

ВОЗВРАТ К СПЕЦПЕРЕХОДУ в общем виде выглядит как **RETLW K**

ВОЗВРАТ К СПЕЦПЕРЕХОДУ в общем виде выглядит как **RETURN**



Спецпереход может быть вложен в другой спецпереход и т.д. не более 8 раз (задействуется т.н. 8-ми уровневый аппаратный стек); как бы то ни было, по команде **RETLW K** мы возвращаемся к команде, следующей за последним спецпереходом.

Команда **RETLW K** делает возврат, вставляя при этом в аккумулятор указанное нами число. Работать с этой парой команд (**CALL** и **RETLW**) нужно очень внимательно, т.к. можно уйти в одно место программы, а вернуться в другом. Пользу от таких телодвижений мы рассмотрим на практике, т.к. переходы – это то, без чего не может существовать ни одна программа.

Альтернативой команде **RETLW K** является команда **RETURN**. Как видно из написания в общем виде, эта команда делает простой возврат без модифицирования содержимого аккумулятора.

Существует также частный случай возврата из спецперехода по команде **RETFIE**, которая кроме возврата осуществляет предварительное разрешение прерываний (прерывания будут рассмотрены далее).

В командах переходов есть частный случай, именуемый как **ВЫЧИСЛЯЕМЫЙ ПЕРЕХОД**. Рассмотрим пример такого перехода.

ВЫЧИСЛЯЕМЫЙ ПЕРЕХОД

```

PCL      EQU      H0002      ; регистр спец назначения
                                ; со счётчиком команд

; ... любая команда

        MOVF      KLOP,W      ; копировать в регистр W число из регистра KLOP

        ADDWF     PCL,F        ; сложить содержимое PCL с содержимым W

; ... команда 1

; ... команда 2

; ... команда 3
  
```

Принцип работы вычисляемого перехода.

Рассматривая метки мы затрагивали работу счетчика команд – это регистр специального назначения **PCL** по адресу h2. Предположим, что в нашей программе 15 строк с командами, которые выполняются друг за другом. После выполнения первой команды счетчик команд автоматически увеличивается на единицу и т.д. до 15-ой команды. В вычисляемом переходе мы можем принудительно изменить счетчик и перейти на другую строчку (как правило, счётчик увеличивают).

Например, в аккумуляторе W у нас число 0, тогда переход произойдет на строчку "команда 1"; а если у нас в аккумуляторе число 2, тогда переход произойдет на строчку "команда 3", т.е. переход на одну команду больше, чем содержимое в аккумуляторе.

Вопросы для самопроверки

Надеюсь, что суть усвоена. К этому моменту должны появиться обоснованные вопросы. Если вопросы не появились, то мы их сформулируем. Далее короткие вопросы-ответы для общего развития и стимула познать больше.

Где в МК расположены строчки с командами?

В памяти программ. Пустые строки без команд не в счёт.

Для чего нужны переходы в программе?

Для организации цикличности программы (о цикличности позже).

Регистр-счётчик вмещает 255 значений, а команд может быть больше, например 1024. Как в этом случае работает счётчик?

Реально счётчик двухбайтный, т.е. используется два регистра. PCL (адрес h2) – младший байт и PCLATH (адрес hA) – старший. Можем менять только младший байт и этого нам достаточно; посчитайте в [BinHexDec](#).

Как работать с отрицательными числами и с числами более 255?

Для работы с такими числами задействуются дополнительные регистры.

Сложение и вычитание мы разобрали, а где умножение деление?

Слишком много вопросов – невозможное возможно.

Команды счётчики

Команды счётчиков используются для упорядоченного увеличения или уменьшения содержимого регистров.

Следует заметить, что рассмотренный нами в вычисляемом переходе регистр PCL – это тоже своеобразный счётчик, но не программный, а аппаратный, тем не менее, значение его регистра можно использовать.

Итак, мы рассмотрим два примера организации счетчиков: первый – конструкция с использованием уже рассмотренных нами команд, а второй – с использованием новых команд.


```
metka1  MOVLW .1      ; записываем в аккумулятор единицу  
  
        ADDWF KLON,F ; увеличиваем (складываем) на единицу  
  
        GOTO metka1 ; переход на метку1
```

Проанализируйте эту конструкцию, она может быть полезна, т.к. увеличивать можно не на единицу, а на любое другое число; уменьшать тоже можно, используя в конструкции команду SUBWF.

В командах счетчиков используют два термина:

- инкрементирование (инкремент), т.е. увеличение регистра на единицу;
- декрементирование (декремент), т.е. уменьшение регистра на единицу.

УВЕЛИЧЕНИЕ НА ЕДИНИЦУ в общем виде выглядит как **INCF F,D**

УМЕНЬШЕНИЕ НА ЕДИНИЦУ в общем виде выглядит как **DECF F,D**

INCF CLON,F ; регистр CLON становится равен CLON+1

DECF CLOP,W ; регистр W = CLOP-1 , регистр CLOP не изменился

При выполнении команды инкрементирования (или декрементирования) с указанием сохранения в аккумулятор, исходные значения регистров после завершения выполнения операций не меняются.

Размер регистра равен 255 – прибавим к 255 единицу и что дальше?

Регистр станет равным нулю – это называется **ПЕРЕНОСОМ**.

А если сложить регистр X равный 250 и регистр Y равный 10?

Регистр станет равным $(X+Y)-255 = 5$. Это тоже перенос.

А если из регистра равным нулю вычесть единицу?

Регистр станет равным 255 – это называется **ЗАЁМ**.

А если из регистра Y равным 10 вычесть регистр X равный 250?

Регистр станет равным $255-(X-Y) = 15$. Это тоже заём.

Рекомендуем вспомнить школьную арифметику и счет столбиком – похожая аналогия.

Переход по событиям в счётчиках

Переход по событиям в счётчиках – ни что иное, как зачатки интеллекта нашего МК. Суть работы заключается в выполнении счёта до определенного момента, т.е. до тех пор, пока регистр, увеличиваемый или уменьшаемый командой счётчиком, не станет равен нулю, а это произойдет либо при "переносе", либо при "заёме".

УВЕЛИЧЕНИЕ НА ЕДИНИЦУ в общем виде выглядит как **INCFSZ F,D**

если после увеличения результат равен 0,

то пропуск следующей команды

УМЕНЬШЕНИЕ НА ЕДИНИЦУ в общем виде выглядит как **DECFSZ F,D**

если после уменьшения результат равен 0,

то пропуск следующей команды

```
MOVLW .255      ; в регистр W записываем число 255
MOVWF CLOP      ; копировать из регистра W в регистр CLOP
INCFSZ CLOP,W    ; увеличиваем на единицу, результат в W
GOTO Metka1      ; переход на Метку1
DECFSZ CLOP,F    ; уменьшаем на единицу, результат в CLOP
GOTO Metka2      ; переход на Метку2
Metka1 ...
Metka2 ...
```

Разберемся с этим фрагментом кода. Сначала мы записали через аккумулятор **W** в регистр **CLOP** число 255, затем его увеличили на единицу (инкрементировали), в результате у нас произошло переполнение (или перенос), т.е. результат операции равен нулю – 0 и как следствие, пропускается команда **GOTO Metka1**. Поскольку мы результат операции увеличения на единицу сохранили в аккумуляторе **W**, то в регистре **CLOP** осталось прежнее число. Уменьшаем регистр **CLOP** на единицу; результат операции 254, и этот результат отличается от нуля, следовательно, выполняется следующая команда **GOTO Metka2**.

Переход по результатам бит-проверки

По сравнению с предыдущими командами это более интеллектуальные команды. Их актуальность во много связана с проверкой состояния ножек МК, а если быть точнее, с проверкой битов в регистрах.

Суть работы – сравнение с единицей или нулем указанного в команде бита. Если значение указанного бита удовлетворяет команде, то выполняется следующая команда, иначе следующая команда пропускается.

Проверить бит на равенство нулю **BTFSC F,B**

если бит B=1, то выполняется следующая инструкция

если бит B=0, то следующая инструкция пропускается

Проверить бит на равенство единице **BTFSS F,B**

если бит B=0, то выполняется следующая инструкция
если бит B=1, то следующая инструкция пропускается

```
    MOVLW b'00011101'; в регистр W записываем число b'00011101'
    MOVWF PORTB      ; копировать из регистра W в регистр PORTB
    BTFSC PORTB,1     ; если 1й бит =1, выполняем след. инструкцию
    GOTO Metka1       ; переход на Метку1
    BTFSS PORTB,1     ; если 1й бит =0, выполняем след. инструкцию
    GOTO Metka2       ; переход на Метку2
Metka1 ...
Metka2 ...
```

Разберем фрагмент этого кода. Помните, биты в регистре нумеруются справа налево и от нуля до семи, следовательно, первый бит в примере у нас равен нулю. Сделав первую проверку по команде **BTFSC PORTB,1** обнаруживается совпадение и пропускается следующая команда – **GOTO Metka1**. Во второй проверке по команде **BTFSS PORTB,1** обнаруживается несовпадение и выполняется следующая команда – **GOTO Metka2**.

Важный вывод. Комбинация команд "переходов по результатам проверки" и команд простых переходов позволяют разветвлять ход программы по нескольким сценариям, тем самым выполнять те или иные задачи.

Флаги как индикаторы событий

Под флагами понимаются значения некоторых битов в регистрах специального назначения. Такие биты в регистрах специального назначения после выполнения определенных команд могут устанавливаться в единицу (иначе говорят – флаг установлен или поднят), а могут опускаться в ноль (иначе – флаг опущен или сброшен).

В регистре STATUS с адресом H03 мы рассмотрим два флага:

– **флаг Z (второй бит)** – флаг нулевого результата

если Z=1, то был нулевой результат выполнения команды

если Z=0, то ненулевой результат выполнения команды

– **флаг C (нулевой бит)** – флаг переноса-займа

если C=1, то было переполнение регистра (происходил перенос)

если C=0, то не было переполнения регистра (либо был заём)

Как работать с флагами?

Если в ходе программы предполагается переполнение регистра (или ожидается заём) и нам необходимо отслеживать эти события, то выполняем следующую последовательность:

```
STATUS    EQU          H0003    ; определяем регистр
```

```

C      EQU      H0000  ; определяем номер бита для флага C
Z      EQU      H0002  ; определяем номер бита для флага Z
;=====

```

; ПРОВЕРКА НА ФАКТ ПЕРЕНОСА

```

...

BCF      STATUS,C ; опускаем флаг C в ноль
;... операции увеличения каких либо регистров

BTFSS    STATUS,C ; делаем бит-проверку C-флага
; если бит C=0, то выполняется следующая инструкция
; если бит C=1, то следующая инструкция пропускается
; =====

Установка флага из нуля в единицу подтверждает факт переноса.

```

; ПРОВЕРКА НА ФАКТ ЗАЙМА

```

...

BSF      STATUS,C ; поднимаем флаг C в единицу
;... операции уменьшения каких либо регистров

BTFSC    STATUS,C ; делаем бит-проверку C-флага
; если бит C=1, то выполняется следующая инструкция
; если бит C=0, то следующая инструкция пропускается
; =====

Установка флага из единицы в ноль подтверждает факт займа.

```

Ранее было обещано рассмотрение команды "MOVF KLOP,F" , в которой результат копирования помещается назад в регистр KLOP.

Если регистр KLOP равен нулю, то после команды "MOVF KLOP,F" флаг Z из нуля поднимется в единицу; иначе из единицы опустится в ноль.

```

;=====

```

; ПРОВЕРКА НА РАВЕНСТВО НУЛЮ

```

CLRf      CLoP      ; обнулим регистр KLoP

BCF       STATUS,Z ; опустим флаг Z в ноль
MOVf      KLoP,F    ; копировать из KLoP в KLoP

BTFSC     STATUS,Z ; делаем бит-проверку Z-флага

```

; если Z=1, то выполняется следующая инструкция, иначе - пропускается

; =====

Умышленная очистка регистра KLoP командой CLRf была сделана ради примера. В результате операции "MOVf KLoP,F" флаг Z из нуля поднимется в единицу, после которого мы делаем проверку и идём по одному из вариантов.

Таким образом, флаги упрощают анализ результатов математических операций и избавляют нас от сложностей, связанных с организацией в программе постоянной проверки содержимого регистров общего назначения.

Команды сравнения

Всё познается в сравнении, в том числе и содержимое регистров. Далее две операции сравнения.

; =====

<p align="center">; ПРОВЕРКА НА РАВЕНСТВО ОДНОГО РЕГИСТРА ДРУГОМУ РЕГИСТРУ в общем виде команда XORWF F,D</p>
--

```

MOVf      CLoP,W    ; копировать из KLoP в W

BCF       STATUS,Z ; опустим флаг Z в ноль
XORWF     CLoN,F    ; проводим сравнение

BTFSC     STATUS,Z ; делаем бит-проверку Z-флага по условию

```

; если Z=1, то выполняется следующая инструкция, иначе пропускается

; =====

Флаг Z после выполнения команды XORWF поднимется из нуля в единицу в том случае, если число в аккумуляторе совпадет с числом, которое находится в проверяемом регистре. Проще говоря, сравнивается содержимое аккумулятора с другим регистром.

; =====

; ПРОВЕРКА НА РАВЕНСТВО
КОНСТАНТЫ КАКОМУ-ЛИБО РЕГИСТРУ
в общем виде команда **XORLW K**

```
MOVF      CLOP,W    ; копировать из KLOP в W  
  
BCF       STATUS,Z  ; опустим флаг Z в ноль  
XORLW     .123       ; проводим сравнение с числом 123  
  
BTFSC     STATUS,Z  ; делаем бит-проверку Z-флага
```

; если Z=1, то выполняется следующая инструкция, иначе - пропускается

; =====

В данном случае сравнивается содержимое аккумулятора с обычным числом (константой).

Команды сдвига битов в регистре

В этих командах используется знакомая вам конструкция **XXXXX F,D**. Рассмотрим теорию работы команд сдвига. Различают две команды – "сдвиг вправо" и, соответственно, "сдвиг влево". Под сдвигом понимается смещение на один бит (на один разряд) содержимого регистра. Закономерен вопрос: а что появляется на освободившемся месте в регистре и куда девается вытесненный бит из регистра? Логично предположить, что место освободившегося бита занимает смещенный бит. Но это не так.

Для этого используется знакомый нам бит C (нулевой бит) в регистре STATUS. Именно через этот бит осуществляется сдвиг содержимого сдвигаемого регистра. На рисунках ниже схематично рассмотрен сдвиг.

Исходное состояние:

бит C (STATUS) =								1
1	0	1	1	0	1	0	0	

Произведен сдвиг вправо:

бит C (STATUS) =								0
1	1	0	1	1	0	1	0	

А теперь сделано два сдвига влево:

бит C (STATUS) =								1
0	1	1	0	1	0	0	1	

Несложно представить ситуацию, когда регистр пуст (все биты равны нулю), а бит C в регистре STATUS

равен единице, то с помощью команды сдвига элементарно просто организовать бегущую единицу, что может найти применение в конструкции бегущего огня.

СДВИГ ВПРАВО битов в регистре RRF F, D

СДВИГ ВЛЕВО битов в регистре RLF F, D

При работе с этой командой необходимо сначала определить состояние бита C в регистре STATUS, а затем выполнять соответствующий сдвиг. Это позволит вам однозначно знать содержимое сдвигаемого регистра после процедуры сдвига.

А теперь для своего блага перепишите на отдельный лист бумаги все рамки с описаниями команд друг за другом. Вам еще не раз потребуется к ним обратиться.

Дорогие друзья, обращаюсь к вам с просьбой. Если у вас есть человеческие примеры и понятные описания использования других команд – прошу не стесняться – вышлите на адрес ntv1978@mail.ru – ваша информация будет отражена на страницах этого проекта.

Всё, что вам не понятно или понятно иным образом, активно обсуждайте на [форуме](#), руководствуясь принципом – "Не бывает глупых вопросов – бывают плохие учителя". Форум сделан специально для этого. Приглашаю всех к обсуждениям!

Глава 3. СОСТАВЛЕНИЕ ТЕКСТА ПРОГРАММЫ НА АССЕМБЛЕРЕ В MPLAB

Текст программы, составленный из команд на языке ассемблер, называется исходным текстом. Исходный текст для МК будем составлять в программе MPLAB. Затем в этой программе исходный текст компилируется, т.е. переводится в машинные коды. Так как наша программа составлена на языке ассемблер, процесс компиляции ещё называют ассемблированием.

Результатом ассемблирования является HEX-файл – обычный текстовый файл с расширением *.hex, который можно открыть стандартным блокнотом Windows. Содержимое этого файла имеет следующий вид:

```
:020000040000FA
:10000000831686018312FF308600092086010920AD
:10001000032855308C008A308D0003308E008C0B05
:0C0020000F288D0B0F288E0B0F280800F6
:02400E00F13F80
:00000001FF
```

Именно такого вида информацию мы будем с помощью нашего программатора прошивать в МК.

Таким образом, в данном разделе мы последовательно рассмотрим:

- установка и подготовка к работе MPLAB;
- создание проекта и подключение файла с программой;
- структуру текста программы;
- правила оформления программы;
- конфигурирование МК;
- особенности сопоставления имен и чисел;
- циклическая концепция программы;
- модульная структура программы;
- понятие о времени исполнения программы;
- задержки в программе и их расчёт;
- компиляция и устранение ошибок.

Установка и подготовка к работе MPLAB

MPLAB IDE – интегрированная среда проектирования. В адрес MPLAB можно сказать много **нехороших** критических высказываний, одно из которых "MPLAB – программа не для людей!". Для удобства работы можно предложить концептуальные усовершенствования, но ввиду того, что MPLAB бесплатен, видимо, это никому не нужно. Не ищите в MPLAB интуитивности в понимании*.

(* Это было первое впечатление. Спустя годы работы я уже ко всему привык и уже все компоненты программы стоят на своих местах. На самом деле, для того чтобы начать работать и получить результат, достаточно знать 4-5 простых действий).

В связи с этим, рекомендую больше осваивать команды ассемблера, а не MPLAB. MPLAB вам покажет явные ошибки. Что касается ошибок организации программы для МК или неправильной логики ее выполнения – это будет на вашей совести; MPLAB здесь не помощник. Тем не менее, в MPLAB присутствуют некоторые инструменты, позволяющие смоделировать и оценить работу программы для МК. Далее мы научимся работе с этими инструментами на конкретных практических примерах.

Какую версию MPLAB ставить? Было проведено сравнение нескольких версий MPLAB: v5.xx, v6.xx, v7.xx. Версии v5.xx – наиболее повернуты лицом к пользователю в плане изучения; в сети Internet имеются русскоязычные описания по работе в версии v5.xx. На описание версий v6.xx не будем тратить время. Версии v7.xx в плане поддержки гораздо хуже, но имеется большой плюс – цветовая подсветка правильности синтаксиса (правильности набора команд). Таким образом, будем работать в версии MPLAB v7.xx и постараемся её развернуть к вам лицом.

Внимание! Версию v7.52 не скачивать; она ставится, но не запускается. У нас стоит v7.50. Скачать MPLAB IDE можно с [официального сайта Microchip](#)** . На другом [дружественном русском](#) сайте можно на начальном этапе обучения поверхностно ознакомиться для общего развития с официально переведенной русскоязычной документацией на тему МК и смежные темы. Ну и конечно всё это (или большая часть) есть [на нашем сайте](#).

(** Это ссылка на архив новых и предыдущих версий программ. Уверен, что вышло множество обновлений. Качайте версию 7.50, т.к. далее в самоучителе примеры именно по этой версии программы, в т.ч. картинки окон MPLAB версии 7.50).

Процесс установки MPLAB IDE прост и не вызывает никаких проблем. Закрываем все открытые посторонние окна и программы. Запускаем инсталлятор, соглашаемся с лицензией, оставляем типичную установку, ставим в предложенную папку, далее... готово.

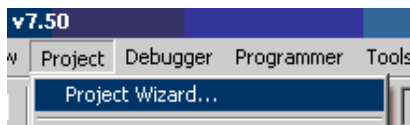
Перед работой создаем отдельную папку Project в месте установки программы, например, здесь C:\Program Files\Microchip\Project\ . В этой папке мы будем создавать и сохранять наши проекты.

Вот и всё.

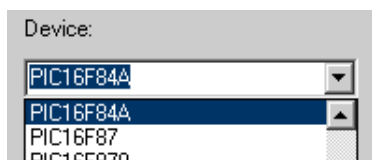
Создание проекта и подключение файла с программой

Как было сказано MPLAB – программа не для людей. Поэтому всё, что будет сказано далее, запоминаем как буквы алфавита.

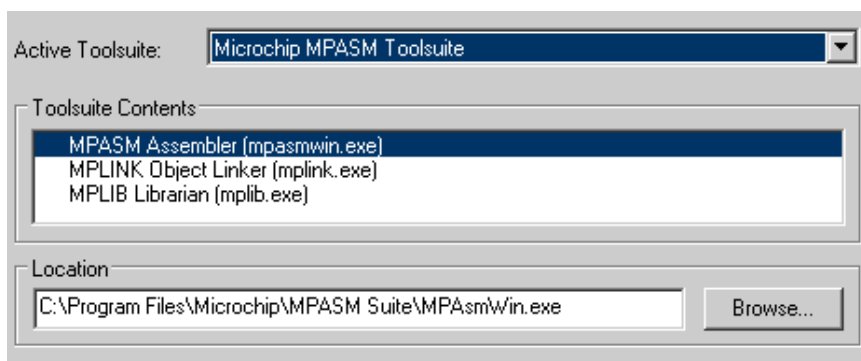
1. Запускаем программу.
2. Запускаем мастер проектов **Project** → **Project Wizard...**



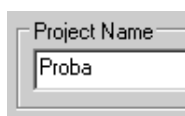
3. Выбираем тип микроконтроллера, например PIC16F84A.



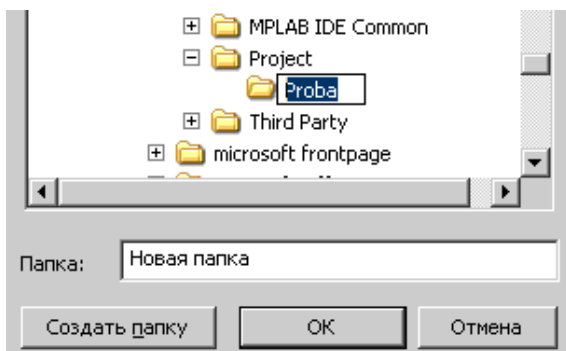
4. Выбор языкового средства, в нашем случае Microchip MPASM Toolsuite и ниже прописываются пути для компилятора, ассемблера, линкера (по умолчанию всё прописано и настроено; перенастраивается в случае работы на других языках, например СИ).



5. Ввод имени проекта (Project Name). Имя проекта строго на английской раскладке без пробелов разумной длины, например, **Proba**.



Выбор папки для размещения проекта (Project Directory). Нажимаем Browse... , находим и выбираем ранее созданную нами папку Project (путь – C:\Program Files\Microchip\Project); далее нажимаем кнопку "Создать папку" которую именуем как **Proba**.



Папку рекомендуем создавать с тем же именем, что и имя проекта.

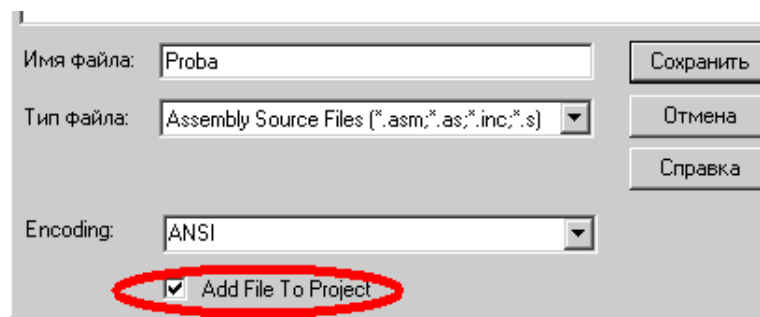
6. Далее, далее, готово.

Проект – это файл с расширением *.mcp который использует MPLAB для своих нужд. Но без этого файла мы не сможем работать. Более того, к этому файлу-проекту нужно подключить наш файл, с текстом программы. Всё это непонятно и запутано, поэтому делаем следующее.

7. Создаем файл, в котором будет набран текст программы. File → New. Появляется пустое окно с заголовком Untitled.

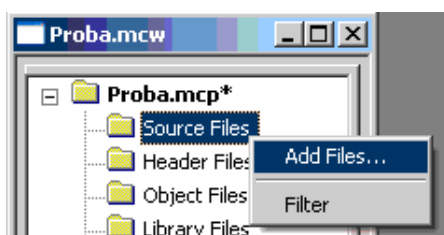


8. Сохраняем этот файл. File → Save As... под именем Proba в папку проекта Proba (путь – C:\Program Files\Microchip\ProjectProba).



А теперь **главное**, перед сохранением поставьте галочку перед **Add File To Project** и этим сделаете **ПОДКЛЮЧЕНИЕ ФАЙЛА С ПРОГРАММОЙ** к проекту.

9. Альтернативное подключение файла с текстом программы (если эту галку забыли поставить) можно сделать иначе.



В окне с заголовком (в нашем примере – Proba.mcw) щелкаем правой кнопкой мыши по Source Files и выбираем пункт **Add Files...**, затем по адресу C:\Program Files\Microchip\ProjectProba находим сохраненный нами файл Proba.asm и открываем его.

Наш файл – Proba.asm – это файл исходник, в котором размещается текст нашей программы на ассемблере. Файл с расширением *.asm, как и файл *.hex, можно открыть в стандартном блокноте Windows. С точки зрения важности – актуален файл исходник, т.к. проект можно создать в любой момент, в любой версии MPLAB и уже потом скомпилировать прошивку.

В ходе продвижения нашего обучения, не смотря на критику, в MPLAB мы покажем некоторые полезные возможности, которые помогут по-новому взглянуть на механизм исполнения программы.

Структура текста программы

Мы установили программу, подключили файл с текстом программы и что дальше? С чего начать? Чем закончить?

Здесь мы рассмотрим структуру текста программы и ответим на вопросы: что, как и в какой последовательности должно располагаться в тексте, а далее рассмотрим нюансы при создании структуры.

Текст программы состоит из двух частей: шапки и тела программы.

В шапке программы указывается:

- тип используемого МК;
- его конфигурация (директива `__CONFIG`);
- перечисляются имена с сопоставленными числами (директива `EQU`).

В шапке программы, образно говоря, определяют правила игры. Действительно, нужно знать какой используется МК, как он настроен (об это позже), какие регистры и какие числа решили сопоставить.

Необходимо отметить, что имена регистров специального назначения мы можем узнать не только из документации, но и в файлах с расширением ***.INC** (путь – C:_Program Files_Microchip_MPASM Suite), например, P16F84A.INC.

Между шапкой и телом программы стоит директива **ORG**, определяющая начальный адрес программы в памяти программ в МК.

Затем идёт тело программы, которое определяет логику работы МК.

В самом конце идёт директива **END**, определяющая конец программы.

Структура программы

	LIST	P=PIC16F84A
	__CONFIG	H3FF1

Шапка программы	W	EQU	H0000
	F	EQU	H0001
	... прочие наименования		
	org	0	; начало программы
Тело программы	... текст программы		
	end		; конец программы

Правила оформления программы

Текст программы традиционно пишется в три колонки от левого края. Каждая колонка занимает 12 знакомест.

Рассмотрим пример

```

LIST          P=PIC16F84A

__CONFIG      H3FF1

PORTB         EQU          H0006

org           0             ; начало программы

metka         clrfs         PORTB

              bsf           PORTB, 7

              goto         metka

              end           ; конец программы

```

Ничего сложного нет.

В первый столбец записываются определяемые имена и метки.

Во второй столбец – команды и директивы.

В третий столбец – имена регистров и числа.

Большой ошибки не будет, если колонки будут шириной 10...14 знакомест; иногда такое происходит в процессе написания текста программы.

Комментарии записываются после символа "точка с запятой" ; . Не ленитесь и не бойтесь писать комментарии, пусть даже самые глупые. Глупый комментарий, как правило, самый запоминающийся.

Для лучшего зрительного восприятия блоки программы можно друг от друга отделять пустыми строками.

Комментарии и пустые строки игнорируются компилятором. Компилятор обрабатывает только тело программы, заключенное между директивами ORG и END.

Конфигурирование МК. Директива __CONFIG

Конфигурирование МК рассмотрим на примере PIC16F84A. Для других типов МК принцип конфигурирования такой же.

Под конфигурированием МК понимается установка битов конфигурации в регистре по адресу H2007. Это необычный регистр – у него не 8 бит, а 14.

Рабочие биты с 0-го по 4-й, не рабочие – с 5-го по 13-й. Не рабочие биты читаются (и записываются) как единица.

13	12	11	10	9	8	7	6	5	4	3	2	1	0
Не задействованы (читаются как 1)									CP	-PWRTE	WDTE	FOSC1	FOSC0
1	1	1	1	1	1	1	1	1	0/1	0/1	0/1	0/1	0/1

Описание конфигурационных битов.

Бит 4 CP бит защиты памяти программ	1 – защита выключена 0 – защита включена
Бит 3 -PWRTE бит разрешения работы таймера включения питания PWRT	0 – будет производиться выдержка при включении питания 1 – выдержки производиться не будет
Бит 2 WDTE бит разрешения работы сторожевого таймера WDT	1 – WDT включен 0 – WDT выключен
Бит 1-0 FOSC1-FOSC0 биты выбора типа генератора	

Описание комбинаций битов выбора генератора.

00	LP – генератор	НЧ генератор для экономичных приложений
01	XT – генератор	Стандартный кварцевый генератор
10	HS – генератор	ВЧ кварцевый генератор
11	RC – генератор	RC генератор с внешней RC цепью

Кратко опишем суть рабочих битов.

Бит защиты памяти программ CP. Используя программатор мы записываем программу в МК. С помощью программатора можно прочитать записанную программу и продублировать её на другом МК. Если установлен бит защиты, то прочитать записанную программу невозможно (защита от копирования). В примерах бит защиты отключен.

Бит разрешения работы таймера включения питания –PWRT. После включения МК требуется некоторое время для стабилизации частоты кварцевого генератора и стабилизации напряжения. С помощью таймера включения осуществляется задержка на время 72мс (типовое время). В примерах таймер включен.

Бит разрешения работы сторожевого таймера WDT. Суть работы сторожевого таймера заключается в периодическом сбросе МК, после чего МК начинает выполнение программы с начала. Используется в тех случаях, когда к устройству на МК предъявляются высокие требования к отказоустойчивости. Используя сторожевой таймер, в текст программы периодически необходимо вставлять команду CLRWDТ – сброс сторожевого таймера. Сторожевой таймер имеет дополнительные настройки связанные с интервалами между сбросами. В примерах сторожевой таймер отключен.

Биты выбора типа генератора OSC. Существуют различные типы генераторов, отличающиеся конструктивным исполнением, способом подключения, частотой генерации и стабильности. В наших примерах мы используем стандартный кварцевый генератор.

Как для директивы `__CONFIG` определить число конфигурации?

Сначала записываем бинарное число в соответствии с нашими настройками:

11111111 – неиспользуемые девять бит в состоянии 1;

1 – защита выключена;

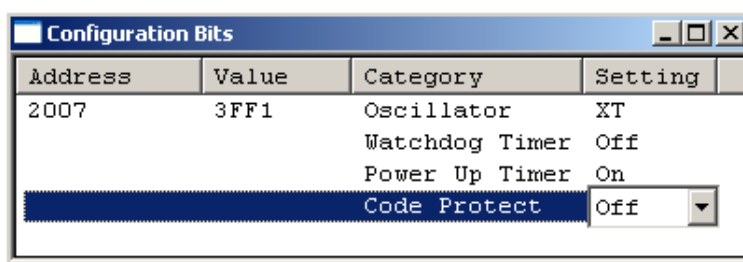
0 – выдержка при включении;

0 – сторожевой таймер отключен;

01 – стандартный кварцевый генератор.

Получается число **1111111110001** которое либо в конвертере [BinHexDec](#), либо в стандартном калькуляторе переводим в 16-тиричное – **h3FF1**. Число, полученное таким образом, указывается после директивы `__CONFIG`.

Шестнадцатеричное число конфигурации можно определить иначе, без использования конвертера или калькулятора. Открываем в MPLAB меню **Configure -> Configuration Bits...** Здесь мы можем выбрать для проекта значения опций конфигурирования МК.



Число в колонке "Value" является искомым числом.

В этом разделе мы также рассмотрим конфигурирование МК путем использования битов конфигурации в явном виде. Биты конфигурации как и регистры специального назначения описаны в файле с расширением *.INC (путь – C:\Program Files_Microchip_MPASM Suite), например, в файле P16F84A.INC . Файл *.INC является обычным текстовым файлом, который можно посмотреть с помощью стандартного текстового редактора "Блокнот" из Windows.

Итак, после строчки с записью LIST P=PIC16F84A необходимо записать две строчки следующего вида

```
#include      P16F84A.INC

__CONFIG _XT_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
```

С помощью #include мы прописываем файл P16F84A.INC с описаниями регистров и битов (с сопоставлениями). Этот файл должен находиться либо в папке MPASM Suite, либо в папке с проектом.

После директивы __CONFIG идёт перечисление соответствующих битов конфигурации. Расшифровку сокращенных наименования битов конфигурации предлагаем вам разобрать самостоятельно.

Какой вариант конфигурирования использовать – это дело личных предпочтений. Мне нравится "короткая" запись типа __CONFIG H3FF1. Длинная запись более наглядна, но требует наличия и прописывания файла типа *.INC. С другой стороны, с прописанным файлом *.INC в шапке программы не требуется сопоставлять регистры специального назначения. В наших дальнейших примерах мы не будем прописывать файл *.INC . Мы будем использовать "короткую" запись для конфигурирования МК и будем вручную сопоставлять регистры специального назначения с тем, чтобы вы обращались к первоисточникам и привыкали к наименованиям регистров специального назначения.

Особенности сопоставления имен и чисел. Прямая и косвенная адресация

Ранее мы рассматривали регистры специального назначения, располагающихся в нулевом банке, но в дальнейшем мы будем не только рассматривать, но и практически использовать регистры специального назначения, расположенные в первом банке памяти, например, TrisA (адрес h85) и TrisB (адрес h86).

Регистрам специального назначения, расположенных в первом банке, назначаются адреса на h80 меньше.

неправильно	правильно
TrisA EQU h85	TrisA EQU h05
TrisB EQU h86	TrisB EQU h06

Этому есть достаточно простое объяснение. Мы знаем, что команда включает слово команды и регистр, указанный в команде. Эту конструкцию мы называем командной строчкой. Командная строчка, таким образом, имеет ширину 14 бит. Выполняя командную строчку, МК для определения адреса регистра использует первые семь бит из команды (берутся из адреса регистра, который указан в команде) и два бита выбора банка (берутся из регистра STATUS – это биты RP1 и RP0). Таким образом получается 9-битный адрес. Проиллюстрируем этот механизм на примере адресации регистра TrisA:

8	7	6	5	4	3	2	1	0
RP1	RP0	Первые семь бит регистра, указанного в команде (TrisA)						
0	1	0	0	0	0	1	0	1

Теперь посмотрим, что у нас получилось: 010000101 = h85.

С этой "заморочкой" разработчиков нужно согласиться и принципом "минус 80" руководствоваться всегда. Это нужно запомнить, в свою очередь мы будем об этом напоминать. На будущее обращаю ваше внимание, регистры TrisA и TrisB получают схожие адреса, что и регистры PortA (адрес h05) и PortB (адрес h06), это также не должно смущать.

В примере выше мы рассмотрели механизм работы "**прямой адресации**". Существует и "**косвенная адресация**", когда обращение к регистрам осуществляется через один и тот же регистр INDF (адрес h00). Для того, чтобы обратиться через этот регистр к какому либо другому регистру (например, PortA), необходимо адрес регистра PortA (число h05) прописать в регистр FSR (адрес h04).

Как определяется 9-битный адрес в команде при работе с косвенной адресацией? В этом случае берутся первые восемь бит из команды и берется бит IRP (7) из регистра STATUS. Если бит IRP установлен в ноль, то будут адресованы регистры с нулевой и первой страниц памяти. Если бит IRP установлен в единицу, то будут адресованы регистры со второй и третьей страниц памяти. PIC16F84A имеет только две страницы памяти, соответственно, использовать этот бит не имеет смысла.

Рассмотрим сегмент очистки 15-ти регистров (диапазон адресов h20 – h2F) с использованием косвенной адресации:

```

                                bcf          STATUS, IRP    ; установить банки 0 и 1

                                movlw        h20             ; указываем первый очищаемый регистр

                                movwf        FSR

m1                               clrf          INDF          ; очистить регистр

                                incf         FSR, F          ; увеличить адрес

                                btfss       FSR, 4           ; проверка

                                goto        m1               ; продолжить очистку

...

```

Сегмент достаточно прокомментирован, однако требует, на наш взгляд, сделать акцент на команде btfss FSR,4 . Используя конвертер [BinHexDec](#) проследим инкрементирование числа h20 до h2F. Если сделать еще один шаг инкрементирования, то 4-й бит в новом числе (h30) установится в единицу. Используя этот факт, мы выходим из сегмента очистки.

Следует отметить, что при использовании прописанного файла *.INC , MPLAB в процессе компиляции будет делать замечание о некорректных адресах регистров специального назначения расположенных в первом банке. Это объясняется тем, что регистры специального назначения из первого банка в файле файл *.INC описаны в традиционном виде, т.е. их адрес будет на h80 больше. При желании файл *.INC можно самим подкорректировать вручную.

Первое знакомство с прямой и косвенной адресацией достаточно ограничить прочтением этого раздела. Большая часть примеров основана на механизме прямой адресации. О сути механизма прямой адресации можно и не знать, однако, могут быть примеры с использованием механизма косвенной адресации, что потребует осмысления этого раздела.

Циклическая концепция программы

Мы уже неоднократно говорили, что команды непрерывно и последовательно выполняются одна за другой. МК не может остановить выполнение программы на какой либо команде, да и команд для остановки и для запуска не существует. Выполнив все строчки с командами, МК не сможет снова их выполнить, если не сделать переход к началу.

Под **циклической концепцией** понимается способность программ непрерывно выполняться. В связи с этим, в более менее серьезных проектах с МК, необходимо тщательно продумывать алгоритм и последовательность выполнения команд.

Нами уже были рассмотрены простейшие примеры заикливания программ с командами переходов. Дабы заинтриговать вас, на практике мы рассмотрим более сложные и более интересные заикливания. В целом вы должны понять, что выполнение программы происходит по "кольцам" с переходами из одного в другое.

Модульная структура программы

Под модульной структурой понимается составление программы из функциональных модулей (кусков, фрагментов, сегментов, подпрограмм). Модули могут выполнять самые разнообразные функции и использовать в самых разнообразных проектах.

При написании программы всегда необходимо стремиться к оптимизации исходного кода – однотипные куски кода оформлять в виде модулей, которые можно в любой момент использовать. В нашем случае мы начнем с простейших примеров, которые выполняют примитивные функции, но, тем не менее, из этих примитивов можно составить более сложные и более функциональные модули.

Другое преимущество модулей – это возможность локализованной отладки. Под локализованной отладкой понимается вынос этого сегмента в отдельный проект, где он обкатывается, обсчитывается, проверяется на работоспособность на демо-плате (об этом далее).

На практике мы научимся базовым приемам использования модулей. Научится этому сможет каждый, а умение оптимизировать и выделять главное придет с опытом.

Понятие о времени исполнения программы

Понятие о времени исполнения программы по важности стоит на втором месте после циклической концепции. Без цикличности не будет работать программа. А без понимания времени выполнения программы нельзя смоделировать правильную работу устройства.

Для начала определимся с цифрами. По нашему опыту скажем, что первоначально у нас очевидные вещи оставались без должного внимания и приводили к ошибочному восприятию скоростных возможностей МК и, как следствие, к ошибкам в программировании.

1 сек = 1000 мс (миллисекунд) = 1000000 мкс (микросекунд)
--

В МК одна команда (точнее один машинный цикл) выполняется за 4 такта опорного генератора.

При частоте кварца в 4МГц на выполнение одной команды уйдет (4 такта / 4000000 Гц) = 1 мкс. Оцените – 1 млн. операций в 1 сек!

С кварцем большей частоты соответственно быстрее.

А теперь смоделируем работу простейшей мигалки. Сначала нам надо установить сигнал на выходе, затем убрать сигнал с выхода, затем зациклить программу. На выполнение этой задачи понадобится около 10 команд. Вопрос – сколько уйдёт времени на выполнение одного полного цикла? Ответ – 10 микросекунд. Это так быстро, что вместо мигания мы получим непрерывное свечение. А если нам нужны фиксированная длительность свечения и фиксированная длительность в выключенном состоянии? Для этого применяются задержки.

Задержки в программе и их расчёт

Задержка – это сегмент программы, который ничего не делает, но обеспечивающий временной простой в ходе выполнения программы на определенное время. Разберем классический пример, обеспечивающий задержку на 10 машинных циклов (10 мкс):

```
Reg_1      equ      H0010

; ...

          movlw      .3
          movwf      Reg_1
wr        decfsz     Reg_1, 1
          goto       wr
```

1я строчка – определяем в "шапке" адрес регистра под число.

2я – записываем число 3 в аккумулятор **W**.

3я – копируем из аккумулятора в **Reg_1**.

Это были подготовительные моменты. Далее собственно сама задержка.

4я строчка – уменьшить **Reg_1** на единицу и результат сохранить **Reg_1**.

5я строчка – переход по метке **wr** на четвертую строчку.

Очевидно, что 4я и 5я строчка в цикле выполняются 2 раза. Тогда где обещанные 10 машинных циклов? Дело в том, что команды, связанные с переходами на другие команды, выполняются за 2 машинных цикла. Теперь всё сходится.

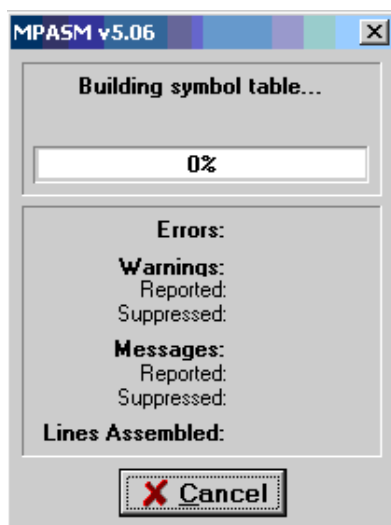
А если нам надо обеспечить задержку не в 10 мкс, а 11 мкс? Тогда достаточно в самом конце кода дописать строчку с командой **NOP** (пустышка), которая как и большинство команд выполняется за 1 мкс.

А как быть если нам нужна задержка для мигалки на 0,5 сек или на 1 сек? Тогда нам надо написать код, который будет занимать 500000 и, соответственно, 1000000 машинных циклов. И это очень просто. Скачиваем программу [Pause_2](#), которая выдаёт код по заданному количеству машинных циклов.

Компиляция и устранение ошибок

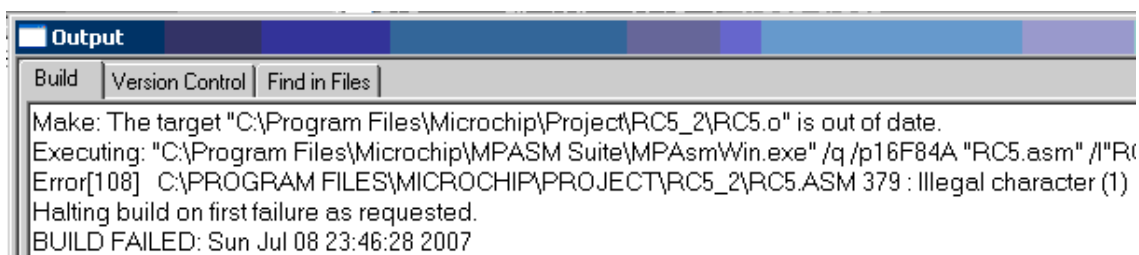
После того, как исходный текст программы написан, его необходимо скомпилировать. В процессе компиляции текст программы переводится в машинные коды.

Для компиляции в MPLAB нажимаем **Project → Make** (или с клавиатуры кнопку **F10**). После нажатия должно появиться окно следующего вида:



В течении 1-2 секунд происходит компиляция исходного текста. Если ошибок в тексте программы нет – то прогресс-полоса с процентами в этом окне зелёного цвета. Если ошибки есть – то прогресс-полоса красного цвета.

В любом случае, по завершении компиляции появляется окно следующего вида:



Мы видим, что в одной из строчек есть запись **Error[108]**. Это значит, что в 108й строке текста программы ошибка. А если дальше прочитать строчку, то можно узнать характер ошибки. Для быстрого поиска указанной строчки с ошибкой достаточно дважды щелкнуть по строке с сообщением об ошибке; будет сделан автоматический переход к ошибочной строчке в тексте программы для МК.

Исправляем и нажимаем F10. Если компиляция проходит без ошибок, то в папке с проектом должен появиться файл с расширением *.hex который мы будем прошивать в МК.

Глава 4. МАКЕТНАЯ ПЛАТА. ПРОГРАММАТОР

В данной главе будут рассмотрены следующие вопросы:

- обозначение ножек микроконтроллера;
- функциональное назначение ножек;
- документация на микроконтроллер PIC16F84A;
- назначение макетной платы;
- схема макетной платы;
- питание макетной платы;
- схема JDM-совместимого программатора;
- инструкция по прошивке микроконтроллера.

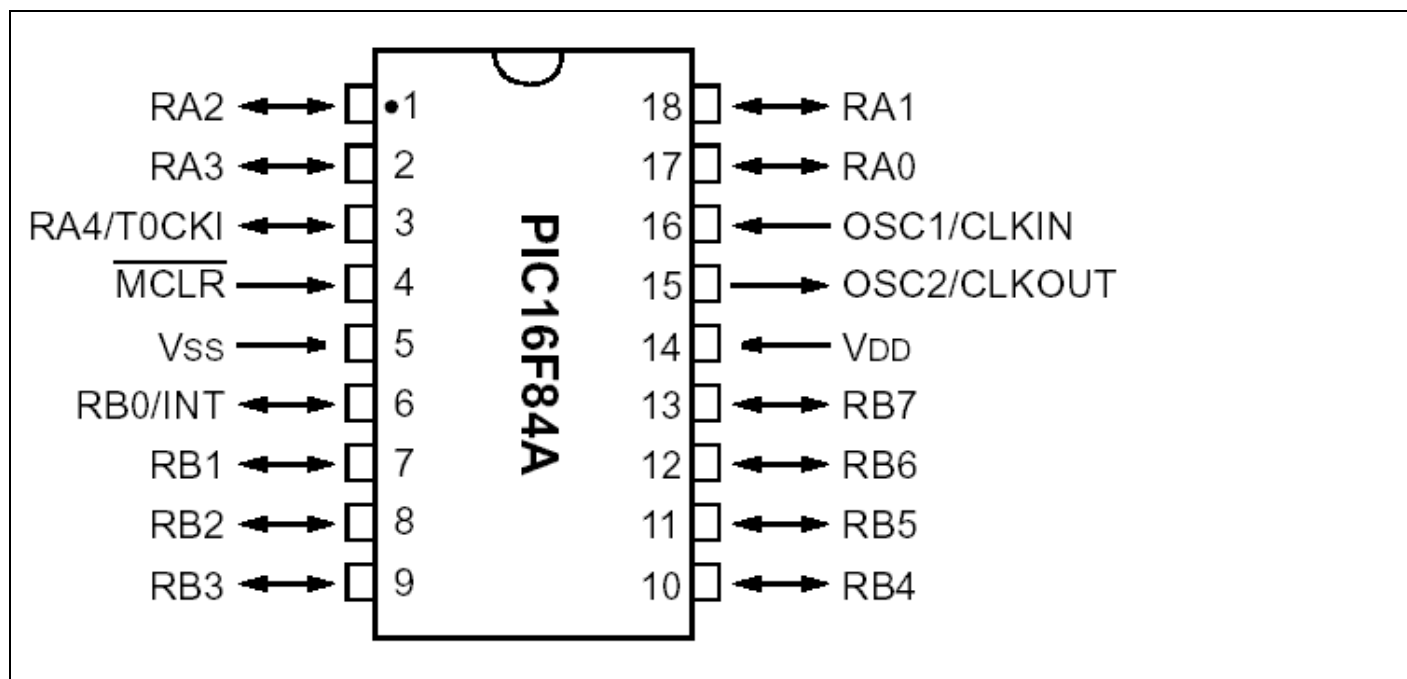
(Отступление. Глядя с высоты полученного опыта, у меня смешанное чувство. С одной стороны я вам

завидую, столько нового и интересного вы узнаете. А с другой стороны, мне вас жалко - не каждому дано освоить железо и его работу. Многие с самого начала подсаживаются на синтетику типа Протеуса. Но это не правильно - нужно хорошо знать работу реального железа).

Обозначение ножек микроконтроллера

Этот раздел было решено поставить первым именно здесь, т.к. набравшись опыта, перед решением задачи целесообразно сначала для себя определиться с чем мы будем работать, и какой МК отвечает нашим задачам. Для этого, как минимум, надо знать, что умеет тот или иной МК и какие ножки отвечают за те или иные функции.

Для описания "ножек" МК в качестве примера мы будем использовать микроконтроллер типа PIC16F84A. Основная задача этого раздела – научить понимать классический набор ножек МК. Вторая задача – показать рисунок МК с описанием ножек для того, что бы вы могли наиболее оптимально разводить рисунок своих будущих печатных плат для конкретных устройств, а лишь затем "под рисунок" составлять свои программы.



Мы видим разнонаправленные стрелки у ножек МК. Направления стрелок указывают нам, как может работать ножка: только на вход (т.е. в МК), только на выход (т.е. из МК), на вход или на выход (в зависимости от настройки).

Также мы видим, что у некоторых ножек обозначения записаны через дробь, например RB0/INT. Это означает, что ножка, в зависимости от настройки может выполнять одну из нескольких функций. В более сложных процессорах, некоторые ножки могут выполнять одну из трёх, или даже четырех функций, например, вход АЦП или выход ШИМ.

Для начала важно чётко различать направление работы ножек.

Направления работы ножек МК и выполняемые функции задаются в регистрах специального назначения.

Этот вопрос рассмотрен в разделе теории и практики работы портов.

Функциональное назначение ножек

Кратко опишем функциональное назначение ножек МК PIC16F84A.

Питание МК. **Vdd (14)** + 5 вольт; **Vcc (5)** – минус питания.

Аппаратный сброс. **MCLR(4)** – микроконтроллер сбрасывается, когда уровень сигнала на этой ножке достигает 0 вольт. Как правило, в схемах эту ножку соединяют с линией +5 вольт.

Опорный генератор. **OSC1(16)** и **OSC2(15)** – для подключения кварцевого резонатора. Также вход **CLKIN(16)** может использоваться для внешнего тактового сигнала, например, для синхронизации работы с другим МК. Выход **CLKOUT(16)** может использовать для вывода тактового сигнала.

Порт А – **RA0, RA1, RA2, RA3, RA4** (соответственно, 17, 18, 1, 2, 3) ножки ввода-вывода. Также ножка 3 может работать как вход частоты для таймера/счетчика TMR0.

Порт В – **RB0, RB1, RB2, RB3, RB4, RB5, RB6, RB7** (соответственно, 6, 7, 8, 9, 10, 11, 12, 13) ножки ввода-вывода. Также 6-я ножка **INT** используется как внешний вход прерывания (о прерываниях позже).

Дополнительными функциями пока не будем забивать голову. Сейчас достаточно знать, что порты в режиме "на выход" могут устанавливать на ножках логические уровни сигналов (0 вольт и +5 вольт), а в режиме "на вход" могут отслеживать установленные на них логические уровни от других устройств, т.е. вести двухсторонний диалог с внешним миром.

Документация на микроконтроллер PIC16F84A

Настала пора дать вам ссылку на описание МК типа PIC16F84A. Полученные ранее знания позволят вам по-новому взглянуть на техническую документацию. Если в документации вы поймете 50% информации – то это большой прогресс. Дальше будет лучше. Но, как правило, поверхностных 50 % бывает достаточно, за исключением узких вопросов. Мы именно так и делаем, акцентируя внимание на нужных периферийных модулях.

К сожалению, на русском языке нет официального перевода документации на PIC16F84A. В связи с этим, для изучения даём два документа. Один на МК типа PIC16F84 на русском языке, второй на МК типа PIC16F84A на английском языке.

[PIC16F84_\(rus\)](#)

[PIC16F84A_\(eng\)](#)

Предлагаем вам найти 17 отличий.

(Дополнительно скачайте и изучайте на русском языке документацию на [PIC16F628A](#))

Макетная плата и её назначение

Макетная плата будет использована нами для закрепления навыков практического программирования под готовое устройство. Будучи "интеллектуальным" лабораторным инструментом, макетная плата сможет найти самое разнообразное применение при тестировании, проведении экспериментов и организации обмена данными.

Кроме этого, в отдельных проектах нашу макетную плату необходимо будет оснастить несложными дополнительными модулями, которые будут рассмотрены в соответствующих проектах.

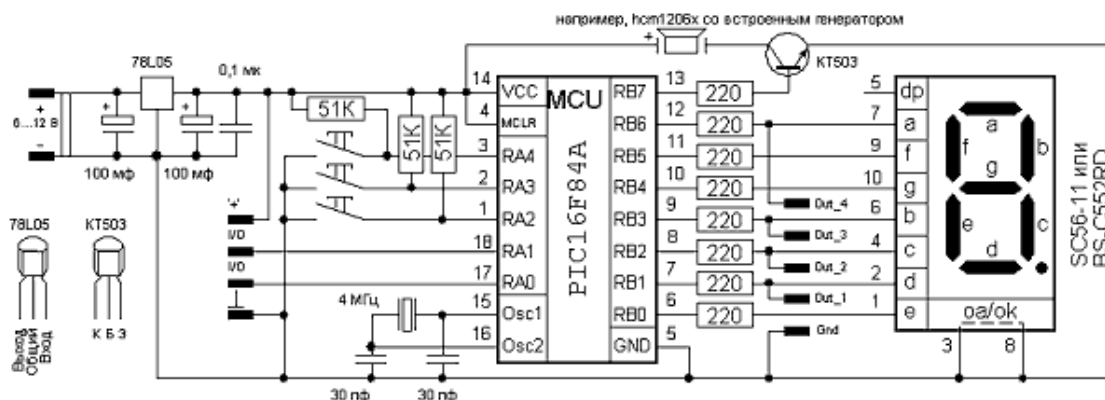
Дополнительно вы можете в несколько шагов за 30 минут [освоить программу Протеус \(Proteus\)](#) для синтетического моделирования ваших проектов. Рекомендую акцент сделать на реальное железо.

Схема макетной платы

Макетная плата и рассматриваемые проекты разработаны под микроконтроллер типа **PIC16F84A**. Это микроконтроллер с флэш-памятью (индекс **F**), что позволяет его гарантированно перепрограммировать 100000 раз, т.е. он **не одноразовый**. Этот МК нельзя назвать "навороченным", но, тем не менее, для обучения и для повторения интересных и полезных проектов его можно и нужно использовать (можно использовать PIC16F628A как совместимый, распространенный и более дешевый).

Схема рисовалась после того, как была разведена печатная плата. Нам гораздо проще сначала оптимально развести навесные элементы под имеющиеся выводы портов, а лишь затем написать программу к "навешенным" элементам. Действительно, нет серьезных препятствий в программе активировать тот или иной вывод порта, а вот компактно и симпатично расположить элементы на плате гораздо сложнее.

Эта схема нам пригодится для программирования МК под имеющиеся электрические соединения.



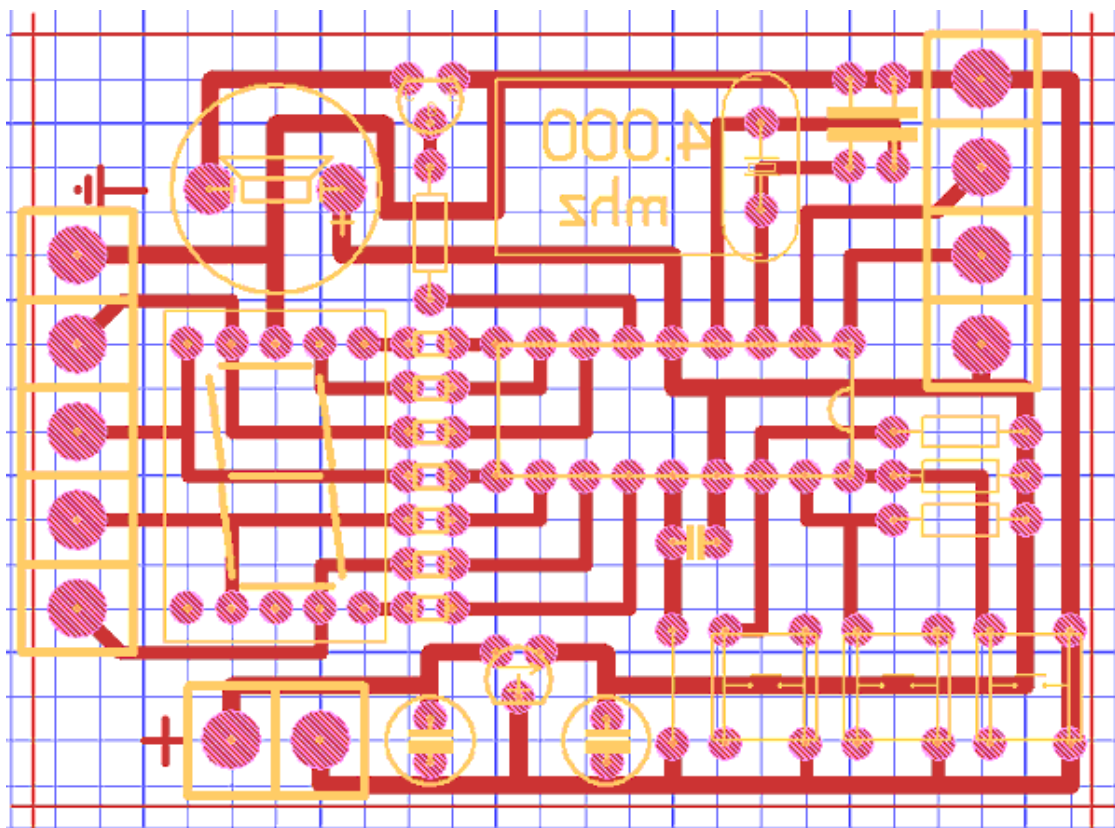
По номиналам детали могут отличаться на $\pm 20\%$. Транзистор любой кремниевый с похожей цоколевкой и проводимостью. Кварц строго на 4 МГц. В макете могут быть использованы сопротивления любых типов, подходящих по габаритам; электролитические конденсаторы общего применения типа К50-6, К50-16 и подобные; неполярные конденсаторы типа КМ, КД, К10-17 и аналогичные. Для коммутирования могут быть использованы тактовые кнопки подходящих посадочных габаритов, например, TS-A1PS-130, TS-A6PS-130 и пр. Электромагнитный излучатель звука со встроенным генератором типа hcm1206x или аналогичный со схожими габаритами.

Мы не советуем использовать другие семисегментные индикаторы с общим катодом, например, типа отечественных АЛС(xxx), т.к. рекомендованный рисунок печатной платы либо потребует изменить, либо в рассмотренных программах придется изменить код, что будет отвлекать внимание от сути. Не забудьте о панели под МК.

Как видно, схема легка в понимании и доступна для повторения, но за этой легкостью и доступностью скрывается великая сила. В макете под будущие проекты запланировано два порта ввода-вывода (**I/O**), сгруппированных с цепями питания. Имеется четыре отдельных порта вывода (**Out**) с общим проводом (**Gnd**). Макет содержит "спартанский" набор навесных элементов: три кнопки, семисегментный индикатор с общим катодом, звуковой излучатель со встроенным генератором (соблюдайте полярность его включения).

Рекомендуем повторить рисунок печатной платы макета, т.к. он тщательно проработан и позволяет легко проследить цепи электрических соединений. (Внимание! Контакты 3 и 8 в семисегментном индикаторе между собой закорочены; на плате электрически соединена только одна ножка).

Вид со стороны печати



Вид со стороны элементов

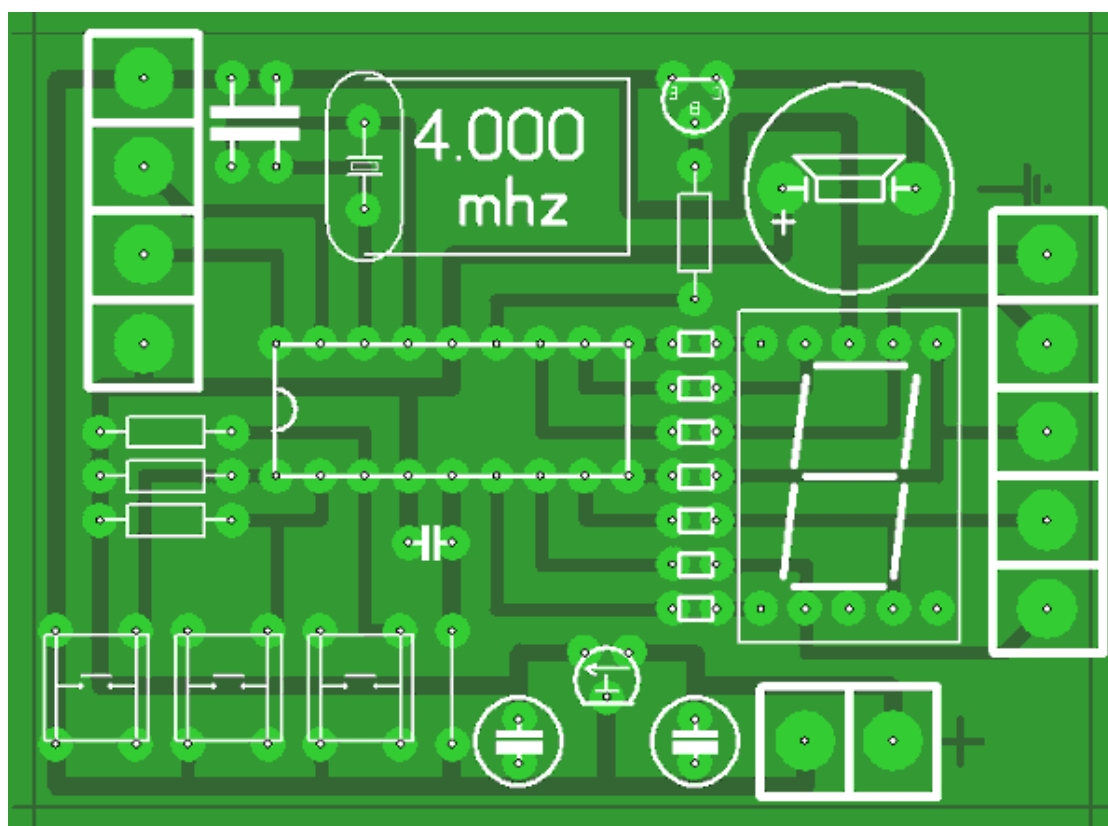
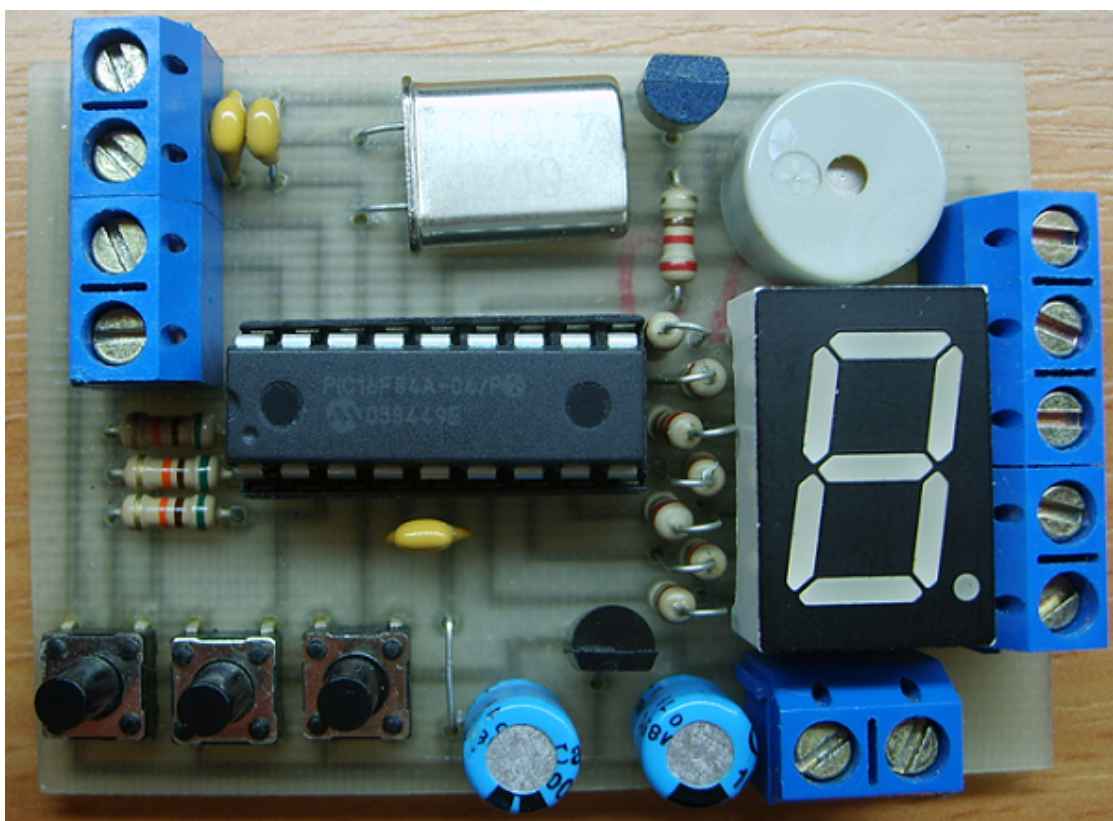
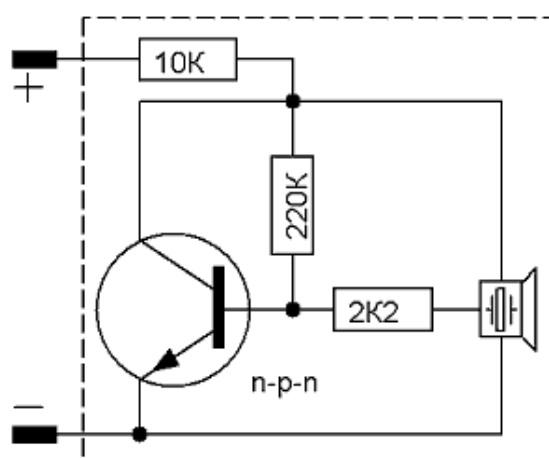


Рисунок печатной платы макета в формате Sprint-Layout можно [скачать здесь](#). Далее фото готовой макетной платы.



Для изготовления печатной платы вы можете воспользоваться лазерно-утюжной технологией (ЛУТ), описание которой [находится здесь](#).

Для справки. Приводим схему бузера (buzzer) – звукового излучателя со встроенным генератором (источник: Радиоаматор, № 10, 2006 г., стр. 34). Трёхвыводной пьезоизлучатель в нашей практике мы не встречали.



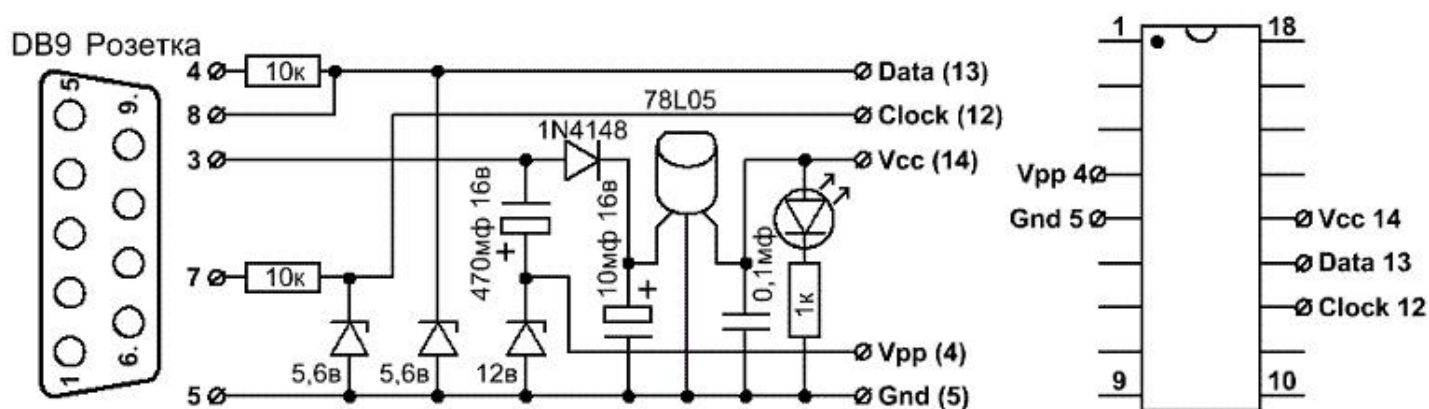
Питание макетной платы

В макетной плате предусмотрен 5-вольтовый стабилизатор напряжения. В связи с этим схему можно подключать к любому блоку питания с постоянным напряжением от 6 до 9 вольт. Можно использовать компактный блок питания, например, от игровой приставки. Не забывайте соблюдать полярность.

В дальнейшем будут рассмотрены альтернативные схемы питания МК.

Схема JDM-совместимого NTV-программатора

В качестве элементарного программатора предлагаем вам собрать по авторской схеме JDM совместимый программатор, который мы назвали NTV программатор. Ниже схема NTV программатора (используется розетка DB9; не путать с вилок).



Собранный по данной схеме программатор многократно и безошибочно прошивал контроллеры PIC16F84, PIC16F84A, PIC16F628A (и ряд других) и может быть рекомендован для повторения начинающим радиолюбителям.

Данный программатор **НЕ РАБОТАЕТ** при подключении к ноутбукам, т.к. уровни сигналов интерфейса RS-232 (COM-порт) в мобильных системах занижены. Также он может не работать на современных ПК, где аппаратно экономится ток на порту. Так что не обессудьте, собирайте и проверяйте на всех попавшихся под руку компьютерах.

Конструктивно плата программатора вставляется между контактами разъема DB-9, которые подпаиваются к контактным площадкам печатной платы. Ниже рисунок платы и фотография собранного программатора.

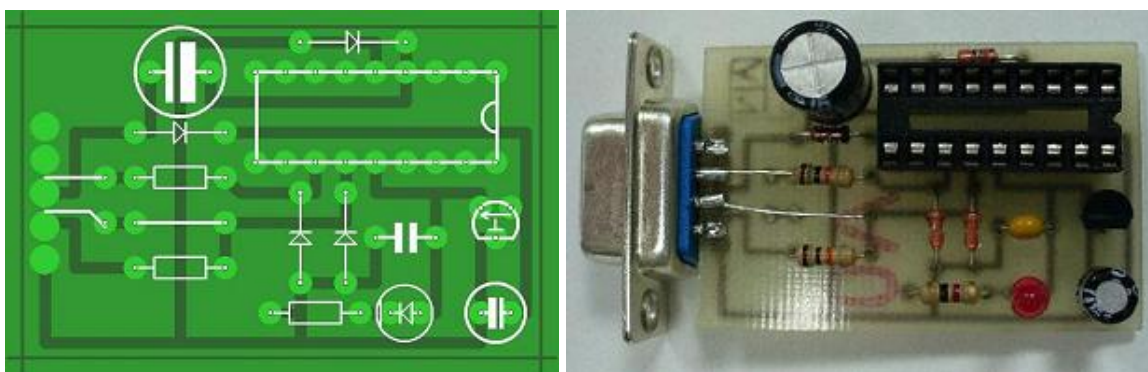


Рисунок печатной платы программатора в формате Sprint-Layout можно [взять здесь](#).

Друзья, если вы имеете хороший навык в конструировании, рекомендуем собрать более [универсальный программатор](#).

Инструкция по прошивке микроконтроллера

1. Соберите JDM-совместимый NTV-программатор, отмойте растворителем или спиртом с зубной щеткой, просушите феном.

Осмотрите на просвет на предмет волосковых замыканий и непропаев. Распаяйте удлинительный шнур мама-папа для COM-порта (не путать с нуль-модемными и кабелями для модемов; прозвоните шнур - первая вилка должна идти к первому гнезду и т.д.; нумерация вилок и гнезд нарисована на самом разъеме).

2. Скачайте программу IC-PROG с [нашего сайта](#) или [с сайта разработчиков](#).

3. Распакуйте программу в отдельный каталог. В образовавшемся каталоге должны находиться три файла:

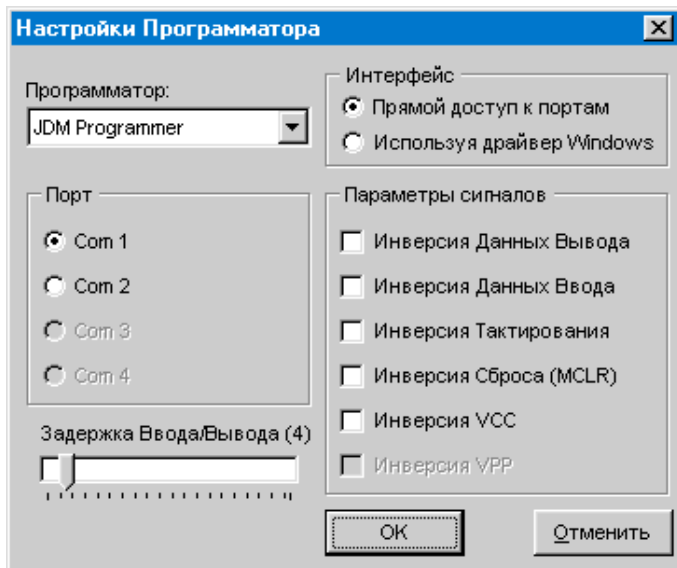
icprog.exe – файл оболочки программатора;

icprog.sys – драйвер, необходимый для работы под Windows NT, 2000, XP. Этот файл всегда должен находиться в каталоге программы;

icprog.chm – файл помощи (Help file).

4. Настройте программу.

Для Windows95, 98, ME	Для Windows NT, 2000, XP
	<p>(Только для Windows XP):</p> <p>Правой кнопкой щёлкните на файле icprog.exe.</p> <p>"Свойства" >> вкладка "Совместимость" >></p> <p>Установите "галочку" на</p> <p>"Запустить программу в режиме совместимости с:" >></p> <p>выберите "Windows 2000".</p>
<p>Запустите файл icprog.exe.</p> <p>Выберите "Settings" >> "Options" >> вкладку "Language" >> установите язык "Russian" и нажмите "Ok".</p> <p>Согласитесь с утверждением "You need to restart IC-Prog now" (нажмите "Ok").</p> <p>Оболочка программатора перезапустится.</p>	
<p>"Настройки" >> "Программатор".</p>	



Проверьте установки, выберите используемый вами СОМ-порт, нажмите "Ok".

Далее, "Настройки" >> "Опции" >> выберите вкладку "Общие" >> установите "галочку" на пункте

"Вкл. NT/2000/XP драйвер" >> Нажмите "Ok" >>

если драйвер до этого не был установлен в системе, в появившемся окне "Confirm" нажмите "Ok". Драйвер установится, и оболочка программатора перезапустится.

Примечание:

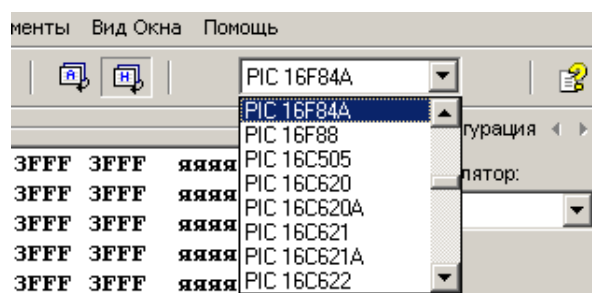
Для очень "быстрых" компьютеров возможно потребуется увеличить параметр "Задержка Ввода/Вывода". Увеличение этого параметра увеличивает надёжность программирования, однако, увеличивается и время, затрачиваемое на программирование микросхемы.

"Настройки" >> "Опции" >> выберите вкладку "I2C" >> установите "галочки" на пунктах:

"Включить MCLR как VCC" и "Включить запись блоками". Нажмите "Ok".

Программа готова к работе.


5. Установите микросхему в панель программатора, соблюдая положение ключа.
6. Подключите шнур удлинителя.
7. Запустите программу IC-Prog.
8. В выпадающем списке выберите контроллер PIC16F84A.




9. Если у вас нет файла с прошивкой – подготовьте его:

- откройте стандартную программу "Блокнот";
- вставьте в документ текст прошивки (со странички УМ-АЦП1);
- сохраните под любым именем, например, proshivka.txt (расширение *.txt или *.hex).

10. Далее в IC-PROG Файл >> Открыть файл (! не путать с Открыть файл данных) >> найти наш файл с прошивкой (если у нас файл с расширением *.txt, то в типе файлов выберите Any File *.*). Окошко "Программного кода" должно заполниться информацией.

11 Нажимаем кнопку "Программировать микросхему" –  (загорается красный светодиод).

12. Ожидаем завершения программирования (около 30 сек.).

13. Для контроля нажимаем "Сравнить микросхему с буфером" – .

Вот и всё. Я тоже думал, что это что-то невероятное. Попробуйте – и у вас получится.

Внимание! Если в тексте программы при конфигурировании МК вы установили бит защиты, то после прошивки появится сообщение об ошибке, т.к. IC-PROG не может прочитать зашитые данные и сравнить их.

Для справки. В 1998 г. на семинаре, организованном фирмой Microchip в Москве, был продемонстрирован вполне легальный способ считывания защищенной прошивки PIC16F84 при помощи российского программатора. Как следствие, через 3 месяца разработчики эту брешь закрыли и выпустили PIC16F84A, одновременно переведя PIC16F84 (без индекса "A") в разряд устаревших.

Глава 5. ЭЛЕМЕНТАРНЫЕ БАЗОВЫЕ ПРОЕКТЫ

В этой главе мы рассмотрим простейшие модели взаимодействия МК с внешними элементами: светодиоды и кнопки. Безусловно, эти модели станут частью более серьезных проектов. Изучив эти проекты, вы поймете принципы модульной организации программ.

Также в этой главе решено рассмотреть работу в МК с энергонезависимой памятью, т.к. она представляет большой практический интерес.

Предполагается осветить следующие вопросы:

- теория и практика работы портов микроконтроллера;
- мигающие светодиоды;
- "бегущий огонь" и "бегущая тень";
- включение символов на индикаторе;
- отслеживание нажатия кнопки;
- кнопка в режиме переключателя и антидребезг;
- работа нескольких копек и многозадачность;
- уменьшение и увеличение значений кнопками;
- энкодер и шаттл: ввод цифровой информации;
- работа с энергонезависимой памятью МК.

Одновременно в этой главе мы расширенно рассмотрим и опишем использование некоторых регистров специального назначения. Разработчики PIC-контроллеров постарались сделать так, что большая часть битов в регистрах специального при включении питания МК по умолчанию находятся в определенном состоянии. Как правило, эти умолчания подходят для выполнения большинства программ. Однако, в некоторых случаях, требуется изменение этих умолчаний.

Теория и практика работы портов МК

Ранее мы рассматривали обозначение ножек МК и упоминали о двунаправленной работе выводов, а именно о работе на вход или на выход. По рисунку несложно определить, что все ножки портов (RA, RB) могут работать и на вход и на выход.

Однако, работа некоторых ножек различается на аппаратном уровне и в связи с этим при разработке схемы устройства следует учитывать эти особенности. Ничего страшного в этом нет, т.к. мы постараемся всё

доходчиво разобрать.

Мы уже знаем, что ножки МК сгруппированы в т.н. порты. Ножки с префиксом RA относятся к порту А, а ножки RB – к порту В. Это свойство группировки позволяет нам работать одновременно с несколькими ножками, т.е. с помощью одной команды за раз можно отследить или изменить состояние всех ножек порта через регистры PORTA и PORTB. С другой стороны, нет никаких препятствий для индивидуальной работы с конкретной ножкой конкретного порта.

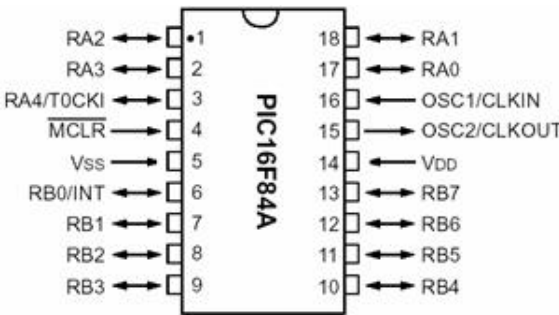
За работу каждого порта в PIC16F84A отвечают по два регистра специального назначения (в скобках указаны соответствующие адреса этих регистров в области оперативной памяти):

Порт А	Порт Б
TRISA (h85) или (h05)	TRISB (h86) или (h06)
PORTA (h05)	PORTB (h06)

Биты в регистрах TRISA и TRISB определяют направление работы каждой ножки МК. По умолчанию все ножки работают на вход (все биты в регистрах установлены в единицу).

Рассмотрим соответствие битов в регистрах TRISA и TRISB реальным ножкам в портах МК на примере PIC16F84A.

бит	7	6	5	4	3	2	1	0
TRISA	–	–	–	RA4 3 выв.	RA3 2 выв.	RA2 1 выв.	RA1 18 выв.	RA0 17 выв.
TRISB	RB7 13 выв.	RB6 12 выв.	RB5 11 выв.	RB4 10 выв.	RB3 9 выв.	RB2 8 выв.	RB1 7 выв.	RB0 6 выв.



Для того, чтобы какая либо ножка работала на выход достаточно в соответствующем регистре установить соответствующий бит в ноль. Не смотря на оговорку об умолчании работы на вход, признаком хорошего тона является явное указание в тексте программы направления работы ножек.

Итак, установка битов регистров TRISA и TRISB в единицу определяет работу ножек на вход; установка битов в ноль – работа ножек на выход. Общее мнемоническое правило в английском языке:

1 (Input) – работа на вход
0 (Output) – работа на выход

Исходя из этого, важно понимать, что в первую очередь в программе мы определяем направление работы

ножек. Кроме этого, выбранное направление мы можем в любой момент времени в программе изменить, если этого требует схема и логика работы разрабатываемого устройства.

Что касается отслеживания сигналов на входе или установки сигналов на выходе, то эти функции выполняются через регистры PORTA и PORTB. Соответствие битов реальным ножкам в портах МК в регистрах PORTA и PORTB точно такое же, как и у регистров TRISA и TRISB.

Разберем работу ножек на вход. В этом режиме работы мы можем программно определить наличие логических уровней сигнала на входе через регистры PORTA и PORTB. В главе 1 мы давали понятие сигнала. Напряжение на ножке близкое к 0 вольт установит соответствующий бит соответствующего порта в ноль. Наоборот, напряжение близкое к 5 вольт установит соответствующий бит соответствующего порта в единицу.

А теперь разберем работу ножек на выход. В этом режиме путем записи единицы в соответствующий бит соответствующего порта мы установим на реальной ножке единичный логический уровень или 5 вольт. И наоборот, записав ноль в соответствующий бит соответствующего порта мы установим на реальной ножке нулевой логический уровень или 0 вольт.

Необходимо отметить, что после определения работы ножки на выход через TRISA или TRISB ножка находится в т.н. неизвестном состоянии, на ней может установиться как логическая единица, так и логический ноль. В связи с этим, после команды определения работы на выход следует поместить команду установки ножки в определенное состояние, которое нам нужно в данный момент времени. Все эти действия в программе мы комментируем как подготовительные моменты.

Концептуальные моменты. Важно понять, что логический сигнал на выходе будет сохраняться сколько угодно долго до тех пор, пока не изменится состояние бита на противоположное. В том случае, если бит установлен в единицу и в него делается попытка записи опять же единицы, перерывов сигнала в момент записи не происходит (тоже самое относится и к нулевому состоянию бита). Смена одного бита порта не влияет на состоянии других битов этого же порта и, следовательно, не влияет на логические уровни на ножках МК.

Теперь осветим особенности работы ножек. Рассмотрим пример с использованием разомкнутой кнопки, которая одним выводом подключена на минусовой провод, а другим выводом к ножке на порту МК. Замкнем контакты кнопки; соответствующий бит устанавливается в логический ноль. Разомкнем контакты и... логическая единица не установится, т.к. уровню логической единицы соответствует напряжение 5 вольт. Для решения этого нюанса используется внешний т.н. подтягивающий резистор, который одним выводом подключен на плюсовой провод, а другим выводом к ножке на порту МК. Номинал сопротивления находится в диапазоне 10...50 ком. Таким образом, через подтягивающий резистор на ножке устанавливается высокий уровень сигнала и, следовательно, бит устанавливается в единицу.

В корпусе МК у каждой ножки порта В имеется небольшая активная нагрузка (около 100мкА) на линию питания, следовательно, внешние подтягивающие резисторы не нужны. Нагрузка на ножке автоматически отключается, если эта ножка запрограммирована на выход.

По умолчанию активная нагрузка отключена. Здесь и далее под умолчанием понимается состояние регистров после включения питания. Лишь регистры энергонезависимой памяти сохраняют свое значение.

Активная нагрузка включается и выключается в регистре OPTION_REG (адрес h81). Активную нагрузку нельзя включить или выключить индивидуально для каждой ножки; управление активной нагрузкой реализовано одновременно для всех ножек. Либо нагрузка у всех ножек, либо ни у одной. В связи с этим подходом к управлению используется всего лишь один седьмой бит RBPU в регистре OPTION_REG.

В работе на выход все ножки всех портов работают одинаково, за исключением ножки RA4 (по причине её многофункциональности). Эта ножка не может устанавливать на своем выходе высокий уровень сигнала. Однако можно подключить к ножке RA4 внешний подтягивающий резистор к положительной линии питания и имитировать сигнал высокого уровня.

В качестве практического примера приведем программу по работе с портами. Сделаем постановку простейшей задачи – на выводах порта В установить различные сигналы, т.е. сигналы высокого и низкого уровня. Конкретизируем задачу: на четырех ножках будет логический ноль, а на других четырех – логическая единица (сигналы статические, т.е. неизменные в своем состоянии). Порядок следования

сигналов для нас не имеет принципиального значения; в нашем примере сигналы установим через один.

```
LIST          P=PIC16F84A
__CONFIG      H3FF1

STATUS        EQU          H0003
PORTB         EQU          H0006
TRISB         EQU          H0006

org           0              ; начало программы

; подготовительные моменты
START         bsf          STATUS, 5      ; переход в Банк 1
              clrf         TRISB
              bcf          STATUS, 5      ; переход назад в Банк 0

; установка сигналов на порту В
              movlw        b01010101
              movwf        PORTB

              goto         START

              end            ; конец программы
```

Напомним последовательность ваших действий по дальнейшей работе с этим текстом программы: в MPLAB создаем проект, там же создаем новый файл и подключаем его к проекту, копируем в него текст этой программы, компилируем и получаем hex-файл ([см. главу 3](#)). Далее прошиваем МК ([инструкция в главе 4](#)).

Результатом компиляции должен стать такой hex-файл (точнее его содержимое или иначе прошивка):

```
:0200000040000FA
:0C000000083168601831255308600000280C
:02400E00F13F80
:000000001FF
```

Примечание. В целях чистоты эксперимента hex-файл компилировался также и в MPLAB версии 5.xx . Результат идентичный.

Кратко рассмотрим работу программы.

Шапка программы в больших пояснениях не нуждается; в ней описаны используемый микроконтроллер, его конфигурация и используемые регистры.

Строчка с директивой `org 0` указывает начальный адрес программы. Если директива `org` отсутствует, то по умолчанию, как правило, программа начинает выполняться с первой строчки программы. Однако, есть исключение для случаев с прерываниями, когда выполнение программы возобновляется со строчки с адресом 4.

Подготовительные моменты касаются определения направлений работы портов. Регистр TRISB находится в первом банке памяти нам необходимо перейти в него, т.к. по умолчанию мы находимся в нулевом банке. PIC16F84A имеет два банка памяти – нулевой и первый. Переход между банками осуществляется путем изменения 5го бита (RP0) в регистре STATUS. Имя бита RP0 мы не стали умышленно сопоставлять с числом 5, т.к., во-первых, вам будет проще понять суть действий, и, во-вторых, эта операция используется не часто (у нас один раз).

В соответствии с поставленной задачей для выводов порта В мы помещаем в аккумулятор число `movlw`

b01010101 которое затем и помещаем в порт В из аккумулятора. Число вида b01010101 может быть записано в другом виде (в другой системе счисления), но мы решили не тратить время на перевод в другую систему счисления. На перспективу, после разбора этого абзаца, для вас будет интересным изменить биты регистра порта В таким образом, чтобы на выводах появилась рациональная комбинация сигналов, которая приведет к отображению на индикаторе осмысленного символа. Для этого возвращаемся к схеме макетной платы, смотрим какие ножки МК управляют соответствующими сегментами индикатора и определяем нужную комбинацию сигналов. Например, для символа «Н» нам нужно записать двоичное число b00111101 (звуковой излучатель не включаем).

Строчка goto START закольцовывает нашу программу. Необходимо отметить, что отсутствие этой строчки в данной программе не приведет к критической ошибке, т.к. наша задача сводится к установке определенной комбинации сигналов и эту установку достаточно выполнить один раз (вспомните концепцию работы ножек). Тем не менее, какая бы у вас задача не ставилась для МК, вашу программу следует закольцовывать в соответствии с циклической концепцией программы.

Последняя строчка с директивой end указывает для MPASM конец программы. Наличие этой строчки обязательно.

Обращаю внимание, пустые строчки в тексте программы служат для лучшего зрительного восприятия. Компилятор MPASM их игнорирует.

Этот раздел может показаться сложным. Однако его понимание составляет суть физиологии работы и является основой для схемотехнических решений с использованием МК. Мы уверены, что настойчивость во многих из вас ускорит освоение этого раздела и подготовит к покорению новых вершин и гораздо активными темпами.

Пример 1. Мигающие светодиоды

В предыдущем разделе мы получили неоценимый опыт работы с ножками порта МК. Однако у многих это вызовет ироничную улыбку. Действительно, в чем польза монотонного включения группы ножек порта.

В этом примере мы оживим работу нашего МК, попеременно устанавливая высокий и низкий уровни сигналов на одних и тех же ножках.

Рассмотрим следующую программу.

```
LIST          P=PIC16F84A
__CONFIG      H3FF1
STATUS        EQU      H0003
PORTB         EQU      H0006
TRISB         EQU      H0006
Reg_1         EQU      H000C
Reg_2         EQU      H000D
Reg_3         EQU      H000E
org           0          ; начало программы
; подготовительные моменты
bsf           STATUS,5    ; переход в Банк 1
clrf          TRISB
bcf           STATUS,5    ; переход назад в Банк 0
; установка сигналов на порту В
m1            movlw      b11111111 ; запись в аккумулятор
movwf         PORTB        ; перенос из аккумулятора в порт
call          Pause        ; переход на метку (с возвратом)
clrf          PORTB        ; "очистка" порта
call          Pause        ; переход на метку (с возвратом)
goto          m1           ; переход на метку (зацикливание)

;delay = 500000 machine cycles
Pause         movlw      .85
movwf         Reg_1
```



```

movlw      .138
movwf      Reg_2
movlw      .3
movwf      Reg_3
wr         decfsz    Reg_1, F
           goto     wr
           decfsz    Reg_2, F
           goto     wr
           decfsz    Reg_3, F
           goto     wr

           return
           end                ; конец программы

```

И собственно результат компиляции – текст прошивки:

```

:020000040000FA
:10000000831686018312FF308600092086010920AD
:10001000032855308C008A308D0003308E008C0B05
:0C0020000F288D0B0F288E0B0F280800F6
:02400E00F13F80
:00000001FF

```

Разберем эту программу. В шапке программы появилось обозначение трех регистров общего назначения как, например, Reg_1 EQU H00C . Закономерен вопрос – почему именно такие имена и такие адреса? Адреса регистров общего назначения согласно документации лежат в диапазоне от h0C до h4F , т.е. всего 68 регистров; первые три регистра из этого диапазона мы именовали для наших дальнейших целей. Имена регистров выбраны по причине рациональности использования как в плане перечисления, так и в плане восприятия. Действительно, проще регистрам дать "бытовые" номера, чем придумывать какие-то имена и путаться в них.

Подготовительные моменты рассмотрены в предыдущей программе.

Далее следует сегмент установки сигналов на порту. Этот сегмент является основным в работе программы. Суть проста. Сначала устанавливается на всех ножках сигнал высокого уровня (2 команды). Затем выдерживается пауза 0,5 сек (светодиоды включены). Затем порт очищается, что приводит к установке сигналов низкого уровня. Снова пауза (светодиоды выключены). И, наконец, переход к началу, что закидывает нашу программу.

Переход к сегменту паузы производится командой call Pause . Эта команда, как мы упоминали ранее в главе 2, используется в паре с одной из двух команд (RETLW и RETURN). Собственно команду RETURN мы видим в конце сегмента паузы.

Сегмент паузы осуществляет задержку в программе длительностью 0,5 сек. Понятие о времени и расчет задержек рассмотрен в главе 3. Мы использовали программу [Pause ver1.2](#) для создания сегмента паузы. После этого несложно догадаться, почему регистрам общего назначения даны такие. С другой стороны, нет никаких препятствий переименовать регистры в созданном сегменте паузы. Нами был выбран путь меньшего сопротивления.

В этом примере важно понять, что к одному и тому же сегменту паузы (сегменту задержки) мы дважды обращались посредством переходов. Однако, возможен частный нерациональный случай, когда такие переходы не используются и после включения и выключения стоит индивидуальный сегмент задержки. Такой решение должно возникнуть лишь в том случае, если длительность свечения и длительность в потушенном состоянии отличаются по времени. Ниже фрагмент программы такого частного случая

```

; установка сигналов на порту B
m1         movlw      b11111111 ; запись в аккумулятор
           movwf      PORTB      ; перенос из аккумулятора в порт
           call       Pause1     ; переход на метку (с возвратом)
           clrf       PORTB      ; "очистка" порта
           call       Pause2     ; переход на метку (с возвратом)
           goto       m1         ; переход на метку (зацикливание)

```

```

;delay = 500000 machine cycles
Pause1      movlw      .85
             movwf      Reg_1
             movlw      .138
             movwf      Reg_2
             movlw      .3
             movwf      Reg_3
wr1          decfsz     Reg_1, F
             goto       wr1
             decfsz     Reg_2, F
             goto       wr1
             decfsz     Reg_3, F
             goto       wr1
             return
;delay = 1000000 machine cycles
Pause2      movlw      .173
             movwf      Reg_1
             movlw      .19
             movwf      Reg_2
             movlw      .6
             movwf      Reg_3
wr2          decfsz     Reg_1, F
             goto       wr2
             decfsz     Reg_2, F
             goto       wr2
             decfsz     Reg_3, F
             goto       wr2
             return

```

Здесь два сегмента паузы помеченные метками Pause1 и Pause2. Для правильности работы сегментов метки wr1 в соответствующих сегментах паузы переименованы в wr1 и wr2. Обратите внимание, в разных сегментах паузы используются одинаковые регистры общего назначения, т.к. в начале каждой паузы эти регистры заполняются своими значениями.

И даже в этом случае, программу можно попытаться упростить.

```

; установка сигналов на порту В

m1          movlw      b11111111 ; запись в аккумулятор
             movwf      PORTB      ; перенос из аккумулятора в порт
             call       Pause1     ; переход на метку (с возвратом)
             clrf       PORTB      ; "очистка" порта
             call       Pause1     ; переход на метку (с возвратом)
             call       Pause1     ; переход на метку (с возвратом)
             goto       m1         ; переход на метку (зацикливание)

```

Действительно, нам ни что не мешает два раза подряд обратиться к одному и тому же сегменту паузы. Экономия места строчек в программном коде и эстетичность очевидна. По этому поводу можно высказать два противоречивых подхода, каждый из которых имеет право на существование. Первый подход заключается в быстром создании неоптимизированного и объемного кода; лишь бы работал. А второй подход – это создание кода, где нет ничего лишнего, где одно вытекает из другого, а другое является частью

еще какого-либо сегмента. И на всё это требуется время для создания программы. Уверяю вас, второй подход гораздо интереснее. Вы по-новому будете смотреть на машинную математику и логику её выполнения. Через некоторое время вы этому научитесь на наших примерах и иначе не сможете работать.

В заключении этого примера предлагаю вам самостоятельно модифицировать программу и исключить команду очистки порта (clrf PORTB). Вместо этого предлагаю "затирать" число в порту В другим числом, отличающимся от нуля. Даю подсказку – вам потребуется пара команд (movlw и movwf). Поэкспериментируйте, вам понравится.

Пример 2. "Бегущий огонь" и "бегущая тень"

На тему этого примера напрашивается несколько решений создания программы.

Можно последовательно устанавливать на определенной ножке (или группе ножек) сигнал высокого уровня, выдерживать паузу, затем устанавливать сигнал низкого уровня, снова выдерживать паузу и затем устанавливать сигнал на других ножках. Весь этот процесс закичивается, что приводит к созданию соответствующего эффекта. Команду установки сигналов низкого уровня можно исключить, чтобы в работе МК исключить случай всех потухших светодиодов (это дело индивидуальных предпочтений). Вариантов последовательности включения светодиодов можно придумать великое множество, была бы фантазия. Рассмотрим сегмент программы (установка сигналов на порту В)

```
; установка сигналов на порту В

m1      movlw      b00000011 ; запись в аккумулятор
        movwf      PORTB      ; перенос из аккумулятора в порт
        call       Pause      ; переход на метку (с возвратом)
        movlw      b00001100 ; запись в аккумулятор
        movwf      PORTB      ; перенос из аккумулятора в порт
        call       Pause      ; переход на метку (с возвратом)
        movlw      b00110000 ; запись в аккумулятор
        movwf      PORTB      ; перенос из аккумулятора в порт
        call       Pause      ; переход на метку (с возвратом)
        movlw      b11000000 ; запись в аккумулятор
        movwf      PORTB      ; перенос из аккумулятора в порт
        call       Pause      ; переход на метку (с возвратом)
        goto       m1         ; переход на метку (зацикливание)
```

Далее прошивка скомпилированной программы, с использованием рассматриваемого сегмента:

```
:020000040000FA
:100000008316860183120330860010200C30860090
:100010001020303086001020C030860010200328C9
:1000200055308C008A308D0003308E008C0B1628E2
```

:0A0030008D0B16288E0B1628080011

:02400E00F13F80

:00000001FF

В нашей макетной плате используется семисегментный индикатор. Использование МК даёт нам неоспоримую возможность в создании оптимального рисунка печатной платы под конкретные элементы. Мы уже упоминали, что именно программу мы пишем под разрабатываемое устройство, поэтому при создании рисунка печатной платы порядок электрических соединений индикатора с МК был для нас не актуален, лишь бы это были соединения с ножками портов (в нашем случае с ножками порта В).

Результат работы индикатора на макете с указанной выше прошивкой не самый симпатичный. Для самостоятельного занятия предлагаю написать вам программу, которая будет включать светодиоды индикатора таким образом, что будет создаваться эффект "бегущего по кругу сегмента"; направление "бега" выберите сами. Эту задачу несложно решить, обратившись к принципиальной схеме нашей макетной платы.

Мы говорили о нескольких решениях реализации "бегущего огня". Второй, принципиально иной подход можно реализовать с помощью команд сдвига битов в регистре. Рассмотрим пример.

```
LIST          P=PIC16F84A
__CONFIG      H3FF1
STATUS        EQU      H0003
PORTB         EQU      H0006
TRISB         EQU      H0006
Reg_1         EQU      H000C
Reg_2         EQU      H000D
Reg_3         EQU      H000E
org           0          ; начало программы
; подготовительные моменты
    bsf        STATUS,5   ; переход в Банк 1
    clrf       TRISB
    bcf        STATUS,5   ; переход назад в Банк 0
    clrf       PORTB      ; очистка порта
    bsf        STATUS,0   ; установка нулевого бита в единицу

; установка сигналов на порту В
m1          rlf        PORTB,1   ; <<< сдвиг в регистре PORTB
            call       Pause      ; <<<
            goto       m1         ; <<< переход на метку (зацикливание)

;delay = 500000 machine cycles
Pause       movlw      .85
            movwf      Reg_1
            movlw      .138
            movwf      Reg_2
            movlw      .3
            movwf      Reg_3
wr          decfsz     Reg_1, F
            goto       wr
            decfsz     Reg_2, F
            goto       wr
            decfsz     Reg_3, F
            goto       wr

            return
end          ; конец программы
```

Далее прошивка

```
:0200000040000FA
:10000000083168601831286010314860D08200528B5
:1000100055308C008A308D0003308E008C0B0E28FA
:0A0020008D0B0E288E0B0E28080031
:02400E00F13F80
:00000001FF
```

Как видим, текст программы намного проще и суть работы заложена в трёх строчках. Следует обратить внимание на дополнительные подготовительные моменты и на расположение метки ml.

Пример 3. Включение символов на индикаторе

Большой практический интерес представляет включение группы светодиодов, которые в совокупности образуют понятные на бытовом уровне символы. Во-первых, это цифры от нуля до девяти. Во-вторых, это некоторые кириллические и латинские буквы. В-третьих, это специфичные символы, например, нижнего подчеркивания "_", знака равно "=" и пр. Наличие таких возможностей в программе определяет совершенство и привлекательность разрабатываемого устройства.

Чтобы повысить уровень вашего интереса к этой теме обратимся к простому примеру из жизни. Многие товары в магазинах бытовой техники имеют встроенные индикаторы, например, стиральные машинки. Почти такая же модель стиральной машинки, но без индикатора, будет стоить на 20-30% дешевле. И разница эта совсем не копеечная. Мы же эти знания получим за сущие копейки.

Опыт и практика полученные в предыдущих примерах позволят вам создать следующую программу. Отрисуем на индикаторе последовательно символы фразы "УРА", затем символ нижнего подчеркивания "_", одновременно сформируем звуковой сигнал и зацклим нашу программу.

Ниже приведен пример такой программы с использованием так называемой таблицы символов. К таблице символов можно обращаться из любого места программы, что очень удобно и практично.

```
LIST P=PIC16F84A
__CONFIG H3FF1
PC EQU H0002
STATUS EQU H0003
PORTB EQU H0006
TRISB EQU H0006
Reg_1 EQU H000C
Reg_2 EQU H000D
Reg_3 EQU H000E
org 0 ; начало программы
; подготовительные моменты
bsf STATUS,5 ; переход в Банк 1
clrf TRISB
bcf STATUS,5 ; переход назад в Банк 0
; отрисовка фразы "УРА_" + звук
ml clrf PORTB ; очистка порта
movlw .10
call TABLE ; переход на метку TABLE
movwf PORTB
```

```

        call      Pause
        movlw    .11
        call      TABLE      ; переход на метку TABLE
        movwf    PORTB
        call      Pause
        movlw    .12
        call      TABLE      ; переход на метку TABLE
        movwf    PORTB
        call      Pause
        movlw    .13
        call      TABLE      ; переход на метку TABLE
        movwf    PORTB
        bsf      PORTB,7      ; звук
        call      Pause
        goto     m1           ; переход на метку (зацикливание)
;=====
TABLE    addwf    PC,F        ; Содержимое счетчика команд PC = PC + W
        retlw    b01101111 ; 0
        retlw    b00001100 ; 1
        retlw    b01011011 ; 2
        retlw    b01011110 ; 3
        retlw    b00111100 ; 4
        retlw    b01110110 ; 5
        retlw    b01110111 ; 6
        retlw    b01001100 ; 7
        retlw    b01111111 ; 8
        retlw    b01111110 ; 9
        retlw    b00111110 ; Y
        retlw    b01111001 ; P
        retlw    b01111101 ; A
        retlw    b00000010 ; _
        retlw    b00110101 ; h
;=====
;delay = 500000 machine cycles
Pause    movlw    .85
        movwf    Reg_1
        movlw    .138
        movwf    Reg_2
        movlw    .3
        movwf    Reg_3
wr        decfsz   Reg_1, F
        goto     wr
        decfsz   Reg_2, F
        goto     wr
        decfsz   Reg_3, F
        goto     wr
        return
        end                               ; конец программы

```

Далее текст прошивки:

```

:0200000040000FA
:10000000083168601831286010A3016208600262078
:100010000B301620860026200C3016208600262065
:100020000D301620860086172620032882076F349D
:100030000C345B345E343C34763477344C347F3467
:100040007E343E3479347D340234353455308C007E
:100050008A308D0003308E008C0B2C288D0B2C28C1
:060060008E0B2C280800A5
:02400E00F13F80
:00000001FF

```

В сегмент таблицы в программе включены символы цифр. Если нам нужны другие символы, мы можем расширить эту таблицу.

Внимательно изучив текст программы – мы увидим однотипность выполняемых действий. В сегменте отрисовки копируем в аккумулятор W число (последовательно 10, 11, 12, 13). Затем отправляемся в сегмент таблицы. С помощью команды `addwf PC,F` мы складываем содержимое аккумулятора с регистром PC (в этом регистре у МК реализован аппаратный счетчик команд). В результате сложения счетчик увеличивается на величину, которая у нас содержится в аккумуляторе W и происходит переход на соответствующую команду в таблице. По команде `getlw bxxxxxxx` мы помещаем в аккумулятор указанное число (bxxxxxxx) и возвращаемся назад в сегмент отрисовки. Указанное число представляет собой комбинацию битов, которое затем записывается в порт и приводит к отображению осмысленного символа.

Строго говоря, поставленную в примере задачу можно было бы решить гораздо проще, без таблицы символов и лишних скачков по тексту программы. Однако, этот пример показывает классический подход к работе с символами. Обоснуем это. Например, работа двух кнопок может выполнять функцию увеличения и уменьшения значения какого либо регистра. Мониторинг значения этого регистра мы можем в автоматическом режиме осуществлять через индикатор. Как видим из этого примера, число в регистре в любой момент времени может принять любое значение и, следовательно, значение непредсказуемо. Но это не препятствие, т.к. это значение будет автоматически обрабатываться и без таблицы символов в этом случае не обойтись.

Важный момент в этом примере заключается в том, что перемещение в таблице происходит за счет принудительного увеличения аппаратного счетчика команд.

Наше устройство станет более интересным и привлекательным если в нем будут задействованы кнопки. Это будет не безликая мигалка-пищалка, а послушное устройство, способное быть управляемым, а самое главное – быть полезным и выполнять необходимые функции.

Пример 4. Отслеживание нажатия кнопки

Рассмотрим следующую программу, которая по нажатии кнопки будет включать индикатор, а после того как кнопка будет отпущена индикатор будет выключен.

```
LIST          P=PIC16F84A
__CONFIG      H3FF1
STATUS        EQU      H0003
TRISA         EQU      H0005
PORTA         EQU      H0005
TRISB         EQU      H0006
PORTB         EQU      H0006
org           0          ; начало программы
; подготовительные моменты
    bsf        STATUS,5   ; переход в Банк 1
    movlw      b00011111
    movwf      TRISA
    clrf       TRISB
    bcf        STATUS,5   ; переход назад в Банк 0
    clrf       PORTB      ; очистка порта
; отслеживание нажатия кнопки
m1          btfsc        PORTA,2   ; бит-проверка ножки RA2
            goto         m1
            movlw        .255
            movwf        PORTB
            clrf         PORTB
            goto         m1
            end          ; конец программы
```

Далее текст прошивки:

```
:0200000040000FA
:1000000083161F3085008601831286010519062894
:08001000FF308600860106287E
:02400E00F13F80
:00000001FF
```

На нашем макете имеется три кнопки. Физически они подключены к линиям RA2, RA3, RA4, которые внешними подтягивающими резисторами соединены с плюсом питания (о подтягивающих резисторах см. "Теория и практика работы портов"). В нашей программе мы решили отслеживать кнопку на линии RA3.

В шапке программы появились описании регистров касающихся работы ножек порта А, а именно TRISA и PORTA.

В разделе подготовительных моментов мы указали для порта А работу всех ножек на вход (а их у нас 5 штук), записав число **b00011111**. Старшие три бита для регистров TRISA и PORTA не являются значащими и, следовательно, с таким же успехом можно записать число **b11111111**. Строго говоря, в данном примере в регистрах TRISA и PORTA нас интересует только второй бит, касающийся работы ножки RA2 и, следовательно, для определения направления работы ножки можно было бы использовать всего лишь одну команду `bsf TRISA,2`.

Перед началом отслеживания кнопки мы очистили порт В и тем самым погасили все сегменты индикатора.

Нажатие кнопки мы отслеживаем с помощью команды `btfs PORTA,2`, которую мы называем командой бит-проверки. Если кнопка не нажата, то на RA2 присутствует высокий уровень сигнала за счет подтягивающего резистора и, соответственно этот факт приводит к выполнению следующей команды – переход по метке `m1`. В этом месте программа закикливается до тех пор, пока не будет сделано нажатие на кнопку. После нажатия кнопки на RA2 устанавливается низкий уровень сигнала, что приводит к изменению хода программы, а именно к пропуску команды перехода по метке `m1`. Далее записывается число в порт, производится включение сегментов индикатора на 1 мкс. Затем порт очищается, индикатор выключается. Затем производится закикливание.

Таким образом, пока нажата кнопка индикатор периодически включается и выключается. За счет высокой скорости работы МК (1 млн. операций в секунду) у нас создается впечатление что индикатор светится, а не мигает. Необходимо отметить, что индикатор светится не на полную яркость и объяснение этому простое – в выключенном состоянии он находится гораздо большее время. Звуковой излучатель на моей макетной плате вообще не подает признаков жизни.

Пример рассматриваемой программы специально упрощен, для того, чтобы вам было проще понять механизм работы. Такой подход пригоден для учебных целей, но на практике следует делать иначе.

Рассмотрим модифицированный фрагмент

```
; отслеживание нажатия кнопки
m1      clr     PORTB
m2      btfs    PORTA,2      ; бит-проверка ножки RA2
        goto    m1
        movlw   .255
        movwf   PORTB
        goto    m2
```

Сначала выполняется очистка порта В. Затем выполняется бит проверка: если кнопка не нажата – переход по метке `m1` для очистки порта и дальнейшее закикливание. Если кнопка нажата – пропускается переход по метке `m1` и выполняется запись числа в порт В. Затем осуществляется переход на очередную бит-проверку и дальнейшее закикливание.

Как видим, процедуры записи в порт и очистки порта у нас разнесены и не накладываются друг на друга, что приводит к полноценной работе ножек порта.

Пример 5. Кнопка в режиме переключателя. Антидребезг

Во многих устройствах требуется работа кнопки в режиме переключателя, т.е. одно нажатие приводит к одному устойчивому состоянию, а следующее нажатие к другому устойчивому состоянию. Сделаем постановку задачи: первым нажатием включить только верхний сегмент индикатора, следующим нажатием – только нижний сегмент.

Рассмотрим программу

```
LIST          P=PIC16F84A
__CONFIG      H3FF1
STATUS       EQU      H0003
PORTA        EQU      H0005
PORTB        EQU      H0006
TRISA        EQU      H0005
TRISB        EQU      H0006
Reg_1        EQU      H000C
Reg_2        EQU      H000D
Reg_3        EQU      H000E
org          0          ; начало программы
; подготовительные моменты
    bsf       STATUS,5   ; переход в Банк 1
    movlw     b00011111
    movwf     TRISA
    clrf      TRISB
    bcf       STATUS,5   ; переход назад в Банк 0
    clrf      PORTB      ; очистка порта
; отслеживание нажатия кнопки
m1          btfsc      PORTA,2   ; бит-проверка ножки RA2
            goto       m1
m2          btfss      PORTA,2   ; бит-проверка ножки RA2
            goto       m2      ; отслеживаем отжатие кнопки
            movlw     b01000000
            movwf     PORTB
            call      Pause
m3          btfsc      PORTA,2   ; бит-проверка ножки RA2
            goto       m3
m4          btfss      PORTA,2   ; бит-проверка ножки RA2
            goto       m4      ; отслеживаем отжатие кнопки
            movlw     b00000010
            movwf     PORTB
            call      Pause
            goto       m1
;delay = 250000 machine cycles
Pause       movlw     .169
            movwf     Reg_1
            movlw     .69
            movwf     Reg_2
            movlw     .2
            movwf     Reg_3
wr          decfsz     Reg_1, F
            goto       wr
            decfsz     Reg_2, F
            goto       wr
            decfsz     Reg_3, F
            goto       wr
            return
end          ; конец программы
```

Далее текст прошивки:

```
:0200000040000FA
:1000000083161F3085008601831286010519062894
:10001000051D082840308600152005190D28051DEE
:100020000F280230860015200628A9308C004530A4
:100030008D0002308E008C0B1B288D0B1B288E0B25
:040040001B28080071
:02400E00F13F80
:00000001FF
```

Это наиболее простой подход к реализации кнопки в режиме переключателя. Однако, программа требует некоторых комментариев.

После нажатия кнопки осуществляется процедура проверки "отжатия" кнопки. Это нужно для того, чтобы "затормозить" ход выполнения программы. Каким бы быстрым и коротким не было наше нажатие на кнопку, программа в любом случае успеет сделать несколько циклов своей работы (1 млн. операций в секунду).

Кроме этого, мы видим вставку сегмента паузы. Пауза длительностью 0,25 сек позволяет нам избежать явление дребезга контактов, которое ведет к ложным срабатываниям. Не сложно предположить, что в секунду мы не сможем сделать больше четырех нажатий на кнопку, поэтому такая длительность оправдана.

Следует отметить, что проверка "отжатия" и наличие паузы для подавления дребезга определяется каждым конкретным случаем. Например, проверка "отжатия" не нужна, если требуется непрерывное увеличение (уменьшение) значения какого либо регистра (аналогия с кнопками громкости на пульте дистанционного управления). Что касается наличия паузы для подавления явления дребезга, то здесь нужно оценивать длительность времени для выполнения дальнейшего кода программы. В большинстве случаев такая пауза не нужна.

Для самостоятельного рассмотрения предлагаю модифицированный фрагмент программы.

```
; отслеживание нажатия кнопки
m1      btfscc    PORTA,2      ; бит-проверка ножки RA2
        goto     m1
m2      btfscc    PORTA,2      ; бит-проверка ножки RA2
        goto     m2
        btfscc    PORTB,6
        goto     m3
        goto     m4
m3      movlw     b00000010
        movwf     PORTB
        call      Pause
        goto     m1
m4      movlw     b01000000
        movwf     PORTB
        call      Pause
        goto     m1
```

Принцип основан на проверке содержимого порта В, и дальнейшего выполнения программы по одному из двух сценариев. Здесь более интеллектуальный подход к работе в отличии от предыдущего примера, где задача решалась прямолинейно.

Пример 6. Работа нескольких кнопок. Многозадачность

В этом примере мы рассмотрим организованную работу трех кнопок. Каждая кнопка будет отвечать за включение одного из символов фразы "УРА". Факт нажатия будем сопровождать звуковым сигналом. Между нажатиями на индикатор будем выводить мигающий символ нижнего подчеркивания.

```
LIST      P=PIC16F84A
__CONFIG  H3FF1
PC        EQU    H0002
STATUS    EQU    H0003
```

```

PORTA      EQU      H0005
PORTB      EQU      H0006
TRISA      EQU      H0005
TRISB      EQU      H0006
Reg_1      EQU      H000C
Reg_2      EQU      H000D
Reg_3      EQU      H000E
org         0          ; начало программы

; подготовительные моменты
    bsf     STATUS,5    ; переход в Банк 1
    movlw   b00011111
    movwf   TRISA
    clrf    TRISB
    bcf     STATUS,5    ; переход назад в Банк 0
    clrf    PORTB       ; очистка порта

; отслеживание нажатия кнопок
m4      movlw      .169
          movwf      Reg_1
          movlw      .69
          movwf      Reg_2
          movlw      .2
          movwf      Reg_3

wr2      btfs      PORTA,2    ; бит-проверка ножки RA2
          goto      m1
          btfs      PORTA,3    ; бит-проверка ножки RA3
          goto      m2
          btfs      PORTA,4    ; бит-проверка ножки RA4
          goto      m3
          decfsz    Reg_1, F
          goto      wr2

          decfsz    Reg_2, F
          goto      wr2
          decfsz    Reg_3, F
          goto      wr2

          btfs      PORTB,1
          goto      m5
          goto      m6
m5      bcf      PORTB,1
          goto      m4
m6      bsf      PORTB,1

          goto      m4

;===== сегмент отрисовки символов
m1      movlw      .0
          call      TABLE      ; переход на метку TABLE
          movwf      PORTB
          call      beep
          goto      m4
m2      movlw      .1
          call      TABLE      ; переход на метку TABLE
          movwf      PORTB
          call      beep
          goto      m4
m3      movlw      .2
          call      TABLE      ; переход на метку TABLE
          movwf      PORTB
          call      beep
          goto      m4

```

```

;===== сегмент работы звукового излучателя
beep      bsf          PORTB, 7
          call         Pause
          clrf         PORTB
          return

;=====
;delay = 250000 machine cycles
Pause     movlw        .169
          movwf        Reg_1
          movlw        .69
          movwf        Reg_2
          movlw        .2
          movwf        Reg_3
wr        decfsz       Reg_1, F
          goto         wr
          decfsz       Reg_2, F
          goto         wr
          decfsz       Reg_3, F
          goto         wr
          return

;=====
TABLE     addwf        PC, F      ; Содержимое счетчика команд PC = PC + W
          retlw        b00111110 ; Y
          retlw        b01111001 ; P
          retlw        b01111101 ; A
          retlw        b00000010 ; _
;=====
          end          ; конец программы

```

Далее текст прошивки:

```

:0200000040000FA
:10000000083161F308500860183128601A9308C007B
:1000100045308D0002308E00051D1F28851D2428C7
:10002000051E29288C0B0C288D0B0C288E0B0C28F8
:1000300086181B281D2886100628861406280030DE
:100040003F2086002E20062801303F2086002E20EB
:10005000062802303F2086002E20062886173220F0
:1000600086010800A9308C0045308D0002308E00DA
:100070008C0B38288D0B38288E0B38280800820707
:080080003E3479347D34023472
:02400E00F13F80
:000000001FF

```

Прокомментируем сегмент отслеживания нажатия кнопок.

Жирным шрифтом выделены строчки формирующие паузу. Эта пауза нам нужна для организации мигания символа нижнего подчеркивания.

Курсивом (после паузы) выделены строчки, которые организуют поочередное включение/выключение нижнего подчеркивания. Этот символ в данном примере отрисовывается "вручную", т.е. мы не обращаемся к таблице символов.

Жирным курсивом выделены строчки связанные с отслеживанием нажатия кнопок.

Важным моментом в данной программе является **организация многозадачности** в цикле. Мы отслеживаем нажатие трех кнопок и мигаем индикатором. Закономерен вопрос – как эти задачи нам распределить во времени? Кнопки нам надо опрашивать как можно быстрее, т.к. нажатие на них может произойти в любой момент. А индикатором надо мигать гораздо реже. Опишем методику, которая была использована для создания сегмента с реализацией многозадачности.

За основу берется код паузы, создаваемый с помощью [Pause ver1.2](#). В центре кода паузы ставятся наиболее приоритетные задачи, а в конце – те задачи, периодичность выполнения которых определяется временем созданного кода паузы. Следует обратить внимание на имена и места расположения меток (в нашем примере wr2 и m4), а также на расположение команд (goto) от которых осуществляется переход к этим меткам.

Пример 7. Уменьшение и увеличение значений кнопками

В этом примере мы рассмотрим как с помощью двух кнопок можно увеличивать или уменьшать значение какого либо регистра. Факт изменения содержимого регистра мы будем отображать на индикаторе.

```

LIST          P=PIC16F84A
__CONFIG      H3FF1
W             EQU      0
F             EQU      1
PC            EQU      H0002
STATUS        EQU      H0003
PORTA         EQU      H0005
PORTB         EQU      H0006
TRISA         EQU      H0005
TRISB         EQU      H0006
C             EQU      0
Z             EQU      2
Reg_1         EQU      H000C
Reg_2         EQU      H000D
Reg_3         EQU      H000E
Reg_4         EQU      H000F      ; регистр под результат
org           0                  ; начало программы

; подготовительные моменты
    bsf        STATUS,5          ; переход в Банк 1
    movlw      b00011111
    movwf      TRISA
    clrf       TRISB
    bcf        STATUS,5          ; переход назад в Банк 0
; отрисовка нуля и подготовка регистра (очистка перед изменением)
    movlw      b01101111
    movwf      PORTB
    clrf       Reg_4
; отслеживание нажатий кнопок
m3      btfs   PORTA,2            ; бит-проверка ножки RA2 - уменьшение
        goto   m1
        btfs   PORTA,3            ; бит-проверка ножки RA3 - увеличение
        goto   m2
        goto   m3                ; зацикливание проверки
; проверка на ноль (на крайнее значение) и уменьшение значения регистра
m1      bcf     STATUS,Z           ; опустим флаг Z в ноль
        movf   Reg_4,F            ; копировать из Reg_4 в Reg_4
        btfs   STATUS,Z           ; делаем бит-проверку Z-флага
; если Z=1, то выполняется следующая инструкция, иначе - пропускается
        goto   m4                ; переходим на отрисовку значения
        decf   Reg_4,F            ; уменьшить значение на 1 и сохранить
        goto   m4
; проверка на 9 (на др. крайнее значение) и увеличение значения регистра
m2      bcf     STATUS,C           ; опускаем флаг C в ноль
        movlw  .247               ; (255-9)+1 = 247 -> W
        addwf  Reg_4,W            ; (Reg_4)+W
        btfs   STATUS,C           ; делаем бит-проверку C-флага
; если бит C=0, то выполняется следующая инструкция
; если бит C=1, то следующая инструкция пропускается
        goto   m5

```

```

m5      goto      m4
m4      incf       Reg_4,F      ; увеличить значение на 1 и сохранить
        movf       Reg_4,W
        call       TABLE
        movwf      PORTB
        call       Pause
        goto       m3
;=====
TABLE    addwf      PC,F        ; Содержимое счетчика команд PC = PC + W
        retlw      b01101111 ; 0
        retlw      b00001100 ; 1
        retlw      b01011011 ; 2
        retlw      b01011110 ; 3
        retlw      b00111100 ; 4
        retlw      b01110110 ; 5
        retlw      b01110111 ; 6
        retlw      b01001100 ; 7
        retlw      b01111111 ; 8
        retlw      b01111110 ; 9
;=====
;delay = 500000 machine cycles
Pause    movlw      .85
        movwf      Reg_1
        movlw      .138
        movwf      Reg_2
        movlw      .3
        movwf      Reg_3
wr        decfsz     Reg_1, F
        goto       wr
        decfsz     Reg_2, F
        goto       wr
        decfsz     Reg_3, F
        goto       wr
        return
        end                ; конец программы

```

Далее текст прошивки:

```

:0200000040000FA
:1000000083161F308500860183126F3086008F01B2
:10001000051D0D28851D1328082803118F080319B5
:100020001A288F031A280310F7300F07031C19280A
:100030001A288F0A0F081F2086002A200828820706
:100040006F340C345B345E343C34763477344C3467
:100050007F347E3455308C008A308D0003308E0022
:0E0060008C0B30288D0B30288E0B30280800BA
:02400E00F13F80
:00000001FF

```

Прокомментируем новые элементы программы.

В шапке программы добавились описания селектора (см. глава 2, "Сложение и вычитание регистров") и описания битов флагов C, Z (см. глава 2, "Флаги как индикаторы событий").

После включения МК содержимое большинства регистров неизвестно (за исключением некоторых умолчаний). Содержимое порта В после включения также неизвестно. В связи с этим на порту В мы выставляем комбинацию сигналов, приводящую к отображению символа нуля – "0". Действительно, мы ведь должны что-то отобразить после включения МК.

Для Reg_4 мы определили роль регистра, в котором будет результат уменьшения/увеличения на

единицу по факту нажатия кнопок. После включения мы его также обнуляем.

Следующий сегмент отслеживания кнопок прокомментирован в предыдущих примерах.

Далее следуют два сегмента, которые проверяют значение Reg_4 на крайние значения. На нашей макетной плате на одном разряде индикатора мы можем отобразить десять символов цифр от "0" до "9". Соответственно, мы будем в первую очередь делать проверку на наличие этих значений в Reg_4. Почему в первую очередь? Т.к. возможна ситуация, когда значение Reg_4 в результате предыдущего увеличения/уменьшения достигло крайнего значения. Строго говоря, после включения Reg_4 мы уже обнулили. Если бы не было организовано такого контроля, то переход за крайние значения приведет к непредсказуемой работе МК.

Проверка на крайние значения организована двумя разными способами. В первом способе реализована проверка значения регистра на ноль. Во втором способе проверка реализована через отслеживание флага переноса-займа. В нашем случае отслеживался перенос из предположения, что если к крайнему значению 9 прибавить 247 произойдет переполнение регистра и произойдет факт переноса. Что и было сделано. Можно было бы вычитать 10 и отслеживать факт займа.

Операции увеличения/уменьшения регистров осуществляются с помощью команд, которые мы называем командами счётчиками.

Пауза в нашей программе осуществляет задержку после вывода символа на индикатор, т.е. для того, чтобы мы успели отпустить кнопку. Однако, если кнопка будет оставаться нажатой, то значение на индикаторе будет динамично увеличиваться/уменьшаться до крайних значений.

Для самостоятельной модификации программы предлагаю решить следующие задачи:

- добавить контроль отжатия кнопок;
- добавить символы крайних значений "H" и "L" (High и Low);
- добавить озвучивание нажатия кнопок;
- добавить повторяющиеся короткие звуковые сигналы в случае установки крайних значений;
- сделать мигающими символы крайних значений "H" и "L";
- задействовать третью кнопку, после нажатия на которую устанавливается среднее значение, например, число 5.

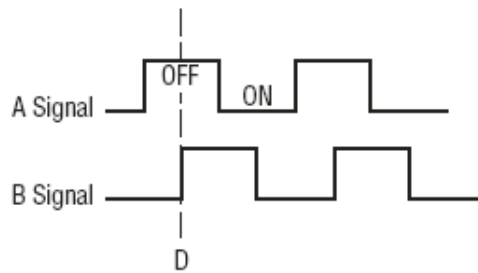
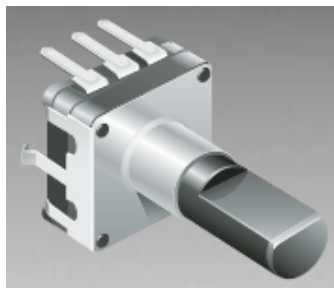
Элементы всех этих задач рассмотрены в предыдущих примерах.

Пример 8. Энкодер и шаттл: ввод цифровой информации

Энкодер иначе называют датчиком вращения (датчиком угла, датчиком поворота). Шаттл – это современное название элемента, который используется для ввода информации и управления цифровой техникой (музыкальные центры, ресиверы, тюнеры, автомагнитолы и пр.). Работа энкодера и шаттла заключается в выводе определенной последовательности импульсов при вращении вала устройства. Далее мы будем придерживаться названия энкодер, т.к. оно употребляется в официальной документации.

Энкодер современный и оригинальный элемент управления цифровыми устройствами. Энкодер по внешнему виду похож на переменный резистор (см. рисунок ниже). Вращение вала сопровождается щелчками, например 24 щелчка на один оборот. Аналогично работает колесо компьютерной мыши, только в данном случае вращают вал вместо колеса. Энкодер имеет 3 вывода – А, В, С и применяется для быстрого ввода данных в цифровые устройства. Некоторые модели имеют встроенную кнопку (например, PEC12-4220F-S0024), которая срабатывает по нажатию на вал энкодера (добавляется еще один вывод). Это свойство позволяет экономить порты МК: нажатием перебирать режимы, а вращением делать настройку. Например, одной ручкой энкодера можно регулировать громкость, баланс, высокие и низкие частоты и пр. режимы, что требует всего лишь трёх сигнальных линий.

Принцип работы. При повороте на один щелчок, например, вправо, сначала замыкается контакт А+С, затем В+С. Когда в этом щелчке вал доворачивается, в той же последовательности контакты размыкаются. При повороте вала в другую сторону, последовательность замыкания с контактом С меняется, т.е. при повороте влево замыкаются сначала В+С, затем А+С.



Приведем пример программы работы с энкодером (например, с использованием энкодера типа PEC12 – Incremental Encoder). Сигнал с вывода А будет использоваться для отслеживания факта вращения, сигнал В – для оценки направления вращения.

```

LIST          P=PIC16F84A
__CONFIG     H3FF1

W    EQU     0
F    EQU     1
PC    EQU     H0002
STATUS EQU     H0003
PORTA EQU     H0005
PORTB EQU     H0006
TRISA EQU     H0005
TRISB EQU     H0006
C    EQU     0
Z    EQU     2
Reg_1 EQU     H000C
Reg_2 EQU     H000D
Reg_3 EQU     H000E
Reg_4 EQU     H000F ; регистр под результат

    org      0      ; начало программы
; подготовительные моменты
    bsf      STATUS,5 ; переход в Банк 1
    movlw    b00011111
    movwf    TRISA
    clrf     TRISB
    bcf      STATUS,5 ; переход назад в Банк 0
; отрисовка нуля и подготовка регистра (очистка перед изменением)
    movlw    b01101111
    movwf    PORTB
    clrf     Reg_4
;
; =====
; отслеживание вращения вала энкодера
En    btfsc   PORTA,0 ; старт от исходного значения (0 или 1)
      goto    en1
      goto    en2
en1    btfsc   PORTA,0 ; выход на единицу
      goto    en1
      goto    en3
en2    btfss   PORTA,0 ; выход на ноль
      goto    en2
      goto    en4
en3    call    Pause ; задержка (антидребезг)
      btfss   PORTA,0 ; проверка на единицу
      goto    En
      goto    en5
en4    call    Pause ; задержка (антидребезг)
      btfsc   PORTA,0 ; проверка на ноль
      goto    En

```



```

        goto     en5
; отслеживание направления вращения
en5      btfsc   PORTA,1
        goto     en6
        goto     en7
en6      call    Pause    ; задержка (антидребезг)
        btfsc   PORTA,1  ; проверка на единицу
        goto     min      ; уменьшение
        goto     En
en7      call    Pause    ; задержка (антидребезг)
        btfss   PORTA,1  ; проверка на ноль
        goto     max      ; увеличение
        goto     En
;=====
; проверка на ноль (на крайнее значение) и уменьшение значения регистра
min      bcf     STATUS,Z  ; опустим флаг Z в ноль
        movf     Reg_4,F   ; копировать из Reg_4 в Reg_4
        btfsc   STATUS,Z  ; делаем бит-проверку Z-флага
        goto     m4        ; переходим на отрисовку значения
        decf     Reg_4,F   ; уменьшить значение на 1 и сохранить
        goto     m4
; проверка на 9 (на др. крайнее значение) и увеличение значения регистра
max      bcf     STATUS,C  ; опускаем флаг C в ноль
        movlw    .247      ; (255-9)+1 = 247 -> W
        addwf    Reg_4,W   ; (Reg_4)+W
        btfss   STATUS,C  ; делаем бит-проверку C-флага
        goto     m5
        goto     m4
m5      incf     Reg_4,F   ; увеличить значение на 1 и сохранить

m4      movf     Reg_4,W
        call     TABLE
        movwf    PORTB
        call     Pause2
        goto     En
;=====
TABLE    addwf    PC,F      ; Содержимое счетчика команд PC = PC + W
        retlw    b01101111 ; 0
        retlw    b00001100 ; 1
        retlw    b01011011 ; 2
        retlw    b01011110 ; 3
        retlw    b00111100 ; 4
        retlw    b01110110 ; 5
        retlw    b01110111 ; 6
        retlw    b01001100 ; 7
        retlw    b01111111 ; 8
        retlw    b01111110 ; 9
;=====
;delay = 100 machine cycles
Pause    movlw    .33
        movwf    Reg_1
wr      decfsz    Reg_1, F
        goto     wr
        return
;delay = 50000 machine cycles
Pause2   movlw    .238
        movwf    Reg_1
        movlw    .65
        movwf    Reg_2
wr2      decfsz    Reg_1, F
        goto     wr2
        decfsz    Reg_2, F

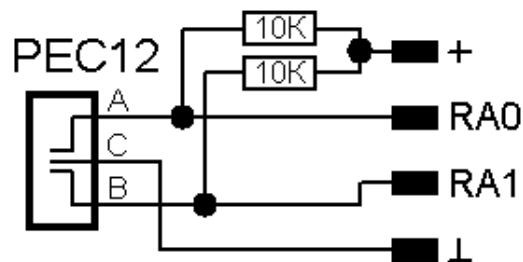
```

```
goto    wr2
return
end      ; конец программы
```

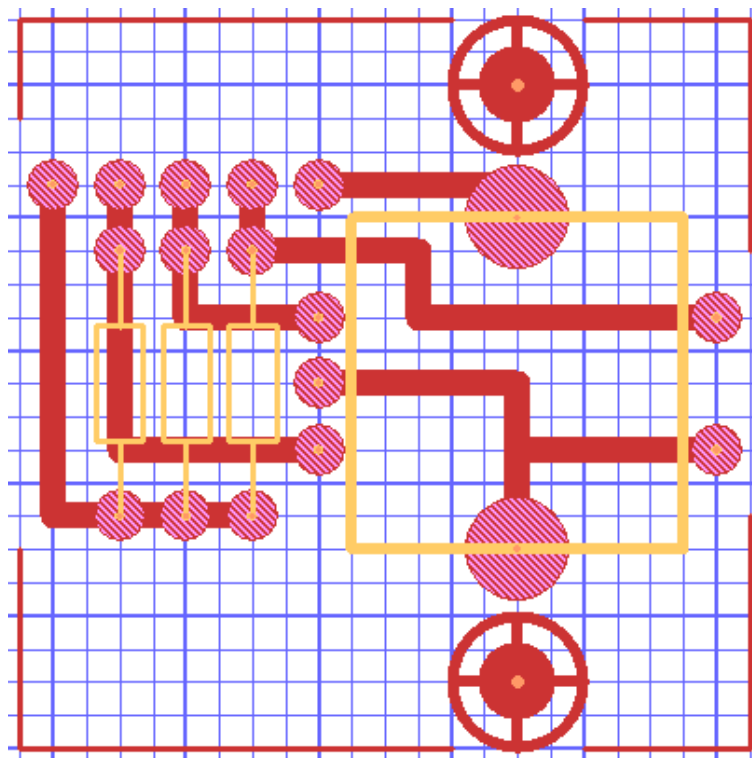
Ниже текст прошивки:

```
:0200000040000FA
:1000000083161F308500860183126F3086008F01B2
:1000100005180B280E2805180B281128051C0E287A
:1000200015284120051C08281928412005180828F2
:10003000192885181C2820284120851824280828DC
:100040004120851C2A28082803118F08031931280C
:100050008F0331280310F7300F07031C3028312895
:100060008F0A0F08362086004620082882076F3442
:100070000C345B345E343C34763477344C347F3427
:100080007E3421308C008C0B43280800EE308C002D
:0E00900041308D008C0B4A288D0B4A28080049
:02400E00F13F80
:00000001FF
```

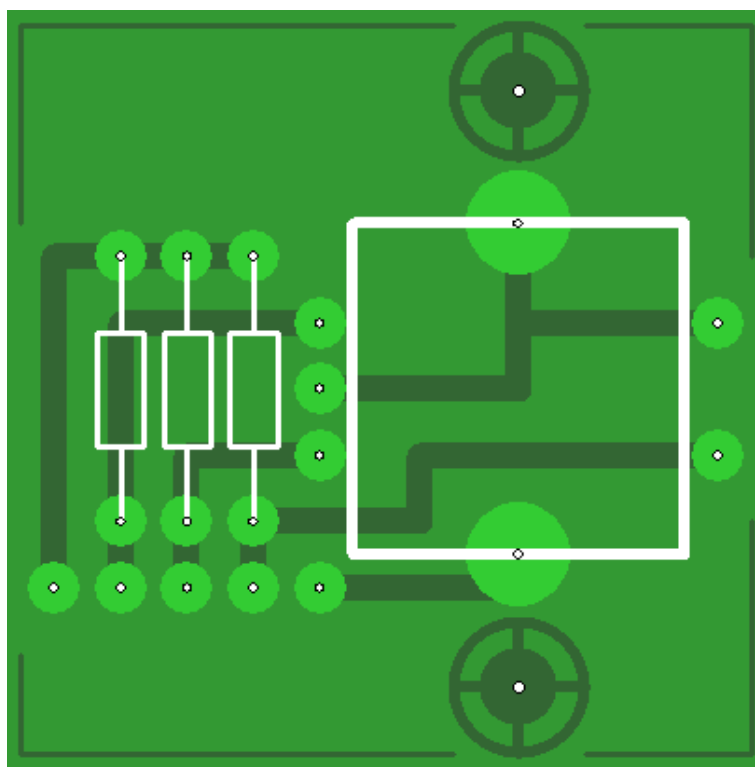
Если предполагается низкая скорость вращения – задержки увеличивают, иначе их можно уменьшить. Задержки для подавления дребезга подбираются экспериментально. Ниже приводим схему подключения энкодера к макетной плате (обозначение элемента нам ранее не встречалось, соответственно, предлагаем свой вариант).



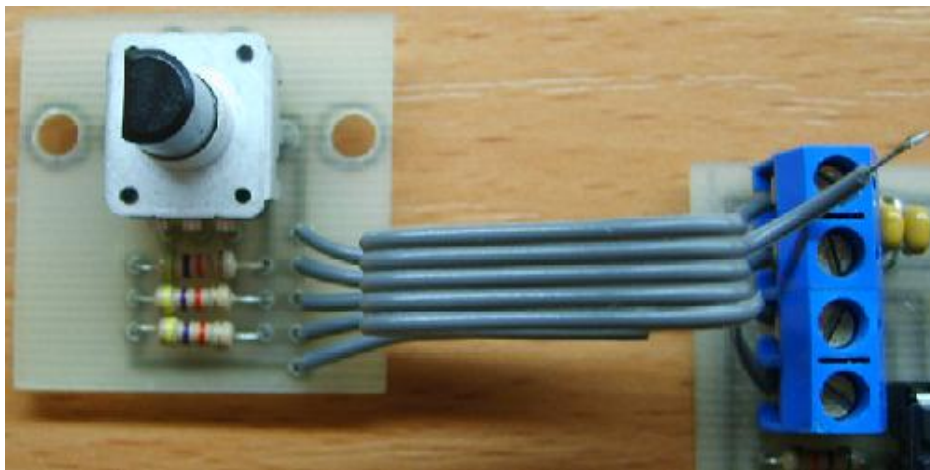
Далее рисунок печатной платы (вид со стороны печати).



Вид со стороны элементов.



Фотография собранного и подключенного к макетной плате энкодера.



На данной плате смонтирован энкодер со встроенной кнопкой. В связи с этим плата разведена с учетом этой особенности на перспективу использования.

Алгоритм работы с энкодером и шаттлом можно использовать для создания устройства, подсчитывающего количество людей, входящих и выходящих в помещение. Алгоритм найдет применение в автоматизации работы турникетов. В качестве датчика можно применить пару простейших оптических приемников и передатчиков.

(В данном примере энкодер обслуживается путем опроса сигнальных линий; по-взрослому энкодер должен обслуживаться по факту прерываний; прерывания будут рассмотрены далее).

Пример 9. Работа с энергонезависимой памятью (ПЗУ)

Энергонезависимая память обладает свойством сохранять свои данные при выключении питания. Примеры реализации этого свойства можно встретить в бытовой технике: продолжение работы стиральной машинки по заданной программе, музыкальные центры сохраняют настройки уровня громкости и тембра, телевизоры сохраняют настройки каналов и т.п.

Также мы предполагаем, что разработчики интеллектуальных устройств на МК используют энергонезависимую память для сознательного ограничения срока работоспособности, тем самым преследуя маркетинговые цели продвижения новых товаров на МК. Нет никаких препятствий для внедрения в программу счетчиков количества включения устройства и/или счетчиков времени непрерывной работы устройства. Искерпав лимит времени устройство в лучшем случае начнет вести себя неадекватно ("глючить"), либо перестанет функционировать вообще.

Далее для простоты восприятия энергонезависимую память в этом тексте мы будем обозначать собирательным термином ПЗУ – постоянное запоминающее устройство. ПЗУ в МК традиционно обозначают фразой EEPROM-память.

Объем ПЗУ не велик: для большинства ПИКов (в том числе и для PIC16F84A) она составляет 64 байта, т.е. 64 регистра, в каждый из которых может быть записан один байт (число от .00 до .255). В дальнейшем значение каждого регистра ПЗУ может быть считано и использовано при исполнении программы. Все регистры ПЗУ имеют свой адрес в диапазоне адресов от .00 до .63 (00h – 3Fh), который используется как при чтении данных из ПЗУ, так и при записи данных в ПЗУ. При записи байта автоматически стирается предыдущее значение и записываются новые данные (стирание перед записью). Все эти операции производит встроенный автомат записи ПЗУ.

Обратите внимание, что числа, определяющие диапазон адресов регистров ПЗУ совпадают с числами диапазона адресов регистров общего и специального назначения (ОЗУ). Это совершенно не должно нас беспокоить, т.к. ПЗУ не принадлежит области регистров ОЗУ.

Существует два варианта записи данных в ПЗУ:

- 1) предварительная запись до начала исполнения программы;
- 2) в ходе исполнения программы.

Далее рассмотрим ключевые сегменты программы, определяющие работу ПЗУ.

```
; текст в шапке программы
; Определение регистров для работы с памятью (операции чтения/записи)
EEData      equ      08h      ; EEPROM – данные
EECon1      equ      08h      ; EECON1 – банк1.
EEAdr       equ      09h      ; EEPROM – адрес
EECon2      equ      09h      ; EECON2 – банк2.
; =====
```

Это сегмент шапки программы. Доступ к ПЗУ осуществляется через два регистра: EEDATA <08h>, который содержит в себе восьмибитовые данные для чтения/записи и EEADR <09h>, который содержит в себе адрес ячейки к которой идет обращение. Дополнительно имеется два управляющих регистра: EECON1 <88h> и EECON2 <89h> (не забываем особенности сопоставления имен и чисел) (вспоминаем формат записи чисел).

```
; ... шапка
; предварительная запись
                org      2100h      ; Обращение к EEPROM памяти данных.
                DE        8h,4h,2h
                DE        1h,3h,5h,7h,8h,6h,4h,2h
; =====
                org      0          ; начало программы
; ... текст программы
```

Этот сегмент программы вставляется между шапкой и рабочей частью программы. Для работы с ПЗУ наличие этого сегмента необязательно, т.к. его задача – предварительная запись определенных значений в ячейки ПЗУ.

Директива org 2100h производит "включение" (разрешение) предварительной записи в ПЗУ.

Далее директива DE 8h,4h,2h произведёт запись трёх чисел в первые три ячейки ПЗУ с адресами 0h, 1h и 2h. В эти ячейки будут записаны числа 8, 4 и 2.

Далее директива DE 1h,3h,5h,7h,8h,6h,4h,2h произведёт запись восьми чисел в следующие восемь ячеек ПЗУ: в третью – 1, четвертую – 3, пятую – 5 ... десятую – 2.

Таким образом, с помощью одной директивы за один раз можно заполнить все 64 регистра ПЗУ, сделав запись в одну строчку через запятую.

Далее традиционный сегмент записи в ПЗУ.

```
; запись данных в энергонезависимую память EEPROM (ПЗУ)
                clrf      INTCON      ; Глобальный запрет прерываний

                movf      Adres,W      ; Записать в регистр W значение Adres
                movwf     EEAdr      ; Скопировать W в регистр EEAdr

                movf      Znach,W      ; Скопировать Znach в регистр W
                movwf     EEData      ; Скопировать W в EEPROM
                bsf       STATUS,RP0   ; Переход в первый банк.
                bsf       EECon1,2     ; Разрешить запись.

                movlw     55h          ; Обязательная
                movwf     EECon2      ; процедура
                movlw     AAh          ; при записи.
                movwf     EECon2      ; ----"----
                bsf       EECon1,1     ; ----"----
                bcf       EECon1,4     ; Сбросить флаг прерывания по окончании
                bcf       STATUS,RP0   ; Переход в нулевой банк.
```

Запрет прерываний – это, упрощенно говоря, действие необходимое для защиты МК от внешних сигналов, которые могут помешать записи данных.

Из регистра Adres мы указываем адрес регистру EEAdr, а из регистра Znach мы копируем наши данные в регистр EEData. Таким образом, мы указываем куда и что записать.

Далее разрешается запись и следуют несколько обязательных строчек, определенных разработчиками Microchip.

Рассмотрим сегмент чтения данных из памяти ПЗУ.

```
; чтение данных из энергонезависимой памяти EEPROM (ПЗУ)
      bcf          STATUS,RP0    ; Переход в нулевой банк.

      movf         Adres,W        ; Записать в регистр W значение Adres
      movwf        EEAdr         ; Скопировать 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 в W из EEPROM
      movwf        Znach         ; Скопировать из W в регистр Znach
```

Из регистра Adres мы указываем адрес регистру EEAdr.

Затем следует процедура чтения после которой в регистр EEData копируется значение из ПЗУ.

Последними двумя строчками копируем значение в регистр Znach.

Теперь работу с памятью рассмотрим на практике.

Используя пример 7 можно поставить и решить две задачи:

- при включении прибора выводить на индикатор последнее значение, которое было перед выключением;
- в ПЗУ хранить данные для счётчика, на основании которых после 5-го включения блокируется работа прибора.

```
LIST          P=PIC16F84A
__CONFIG      H3FF1
W             EQU      0
F             EQU      1
PC            EQU      H0002
STATUS        EQU      H0003
PORTA         EQU      H0005
PORTB         EQU      H0006
TRISA         EQU      H0005
TRISB         EQU      H0006
INTCON        EQU      H000B
C             EQU      0
Z             EQU      2
Reg_1         EQU      H000C
Reg_2         EQU      H000D
Reg_3         EQU      H000E
Reg_4         EQU      H000F    ; регистр под результат
; Определение регистров для работы с памятью (операции чтения/записи)
EEData        equ       08h      ; EEPROM - данные
EECon1        equ       08h      ; EECON1 - банк1.
EEAdr         equ       09h      ; EEPROM - адрес
EECon2        equ       09h      ; EECON2 - банк2.
org          2100h    ; Обращение к EEPROM памяти данных.
DE          5        ; предварительная запись
org           0         ; начало программы
```

```

; подготовительные моменты
    bsf        STATUS,5      ; переход в Банк 1
    movlw     b00011111
    movwf     TRISA
    clrf      TRISB
    bcf       STATUS,5      ; переход назад в Банк 0
; чтение значения из памяти и отрисовка
    call      Read          ; чтение числа
    movf      Reg_4,W
    call      TABLE
    movwf     PORTB
; отслеживание нажатий кнопок
m3      btfss   PORTA,2      ; бит-проверка ножки RA2 - уменьшение
        goto    m1
        btfss   PORTA,3      ; бит-проверка ножки RA3 - увеличение
        goto    m2
        goto    m3          ; зацикливание проверки
; проверка на ноль (на крайнее значение) и уменьшение значения регистра
m1      bcf     STATUS,Z      ; опустим флаг Z в ноль
        movf    Reg_4,F      ; копировать из Reg_4 в Reg_4
        btfsc   STATUS,Z      ; делаем бит-проверку Z-флага
; если Z=1, то выполняется следующая инструкция, иначе - пропускается
        goto    m4          ; переходим на отрисовку значения
        decf    Reg_4,F      ; уменьшить значение на 1 и сохранить
        goto    m4
; проверка на 9 (на др. крайнее значение) и увеличение значения регистра
m2      bcf     STATUS,C      ; опускаем флаг C в ноль
        movlw   .247         ; (255-9)+1 = 247 -> W
        addwf   Reg_4,W      ; (Reg_4)+W
        btfss   STATUS,C      ; делаем бит-проверку C-флага
; если бит C=0, то выполняется следующая инструкция
; если бит C=1, то следующая инструкция пропускается
        goto    m5
        goto    m4
m5      incf     Reg_4,F      ; увеличить значение на 1 и сохранить
m4      movf     Reg_4,W
        call     TABLE
        movwf    PORTB
        call     Pause
        call     Write      ; запись числа
        goto    m3
;=====
TABLE    addwf    PC,F        ; Содержимое счетчика команд PC = PC + W
        retlw    b01101111 ; 0
        retlw    b00001100 ; 1
        retlw    b01011011 ; 2
        retlw    b01011110 ; 3
        retlw    b00111100 ; 4
        retlw    b01110110 ; 5
        retlw    b01110111 ; 6
        retlw    b01001100 ; 7
        retlw    b01111111 ; 8
        retlw    b01111110 ; 9
;=====
;delay = 500000 machine cycles
Pause    movlw   .85
        movwf    Reg_1
        movlw   .138
        movwf    Reg_2
        movlw   .3
        movwf    Reg_3
wr      decfsz   Reg_1, F

```

```

        goto      wr
        decfsz    Reg_2, F
        goto      wr
        decfsz    Reg_3, F
        goto      wr
        return

;=====
; запись данных в энергонезависимую память EEPROM (ПЗУ)
Write    clrf      INTCON      ; Глобальный запрет прерываний
        clrw      ; обнулить W, т.е. установить адрес "0"
        movwf     EEAddr      ; Скопировать W в регистр EEAddr
        movf      Reg_4,W      ; Скопировать данные из Reg_4 в регистр
W
        movwf     EEData      ; Скопировать W в EEPROM
        bsf       STATUS,5     ; Переход в первый банк.
        bsf       EECon1,2     ; Разрешить запись.
        movlw     55h          ; Обязательная
        movwf     EECon2      ; процедура
        movlw     AAh          ; при записи.
        movwf     EECon2      ; ----"----
        bsf       EECon1,1     ; ----"----
        bcf       EECon1,4     ; Сбросить флаг прерывания по окончании
        bcf       STATUS,5     ; Переход в нулевой банк.
        return

;=====
; чтение данных из энергонезависимой памяти EEPROM (ПЗУ)
Read     bcf       STATUS,5     ; Переход в нулевой банк.
        clrw      ; обнулить W, т.е. установить адрес "0"
        movwf     EEAddr      ; Скопировать W в регистр EEAddr
        bsf       STATUS,5     ; Переход в первый банк.
        bsf       EECon1,0     ; Инициализировать чтение.
        bcf       STATUS,5     ; Переход в нулевой банк.
        movf      EEData,W     ; Скопировать в W из EEPROM
        movwf     Reg_4        ; Скопировать из W в регистр Reg_4
        return

;=====
        end                ; конец программы

```

Далее текст прошивки:

```

:0200000040000FA
:10000000083161F3085008601831248200F082120A7
:100010008600051D0E28851D1428092803118F0848
:1000200003191B288F031B280310F7300F07031C2D
:100030001A281B288F0A0F08212086002C2039201F
:10004000092882076F340C345B345E343C347634D8
:1000500077344C347F347E3455308C008A308D00B8
:1000600003308E008C0B32288D0B32288E0B3228F9
:1000700008008B01030189000F088800831608150A
:1000800055308900AA3089008814081283120800AC
:1000900083120301890083160814831208088F0055
:0200A000080056
:02400E00F13F80
:024200000500B7
:00000001FF

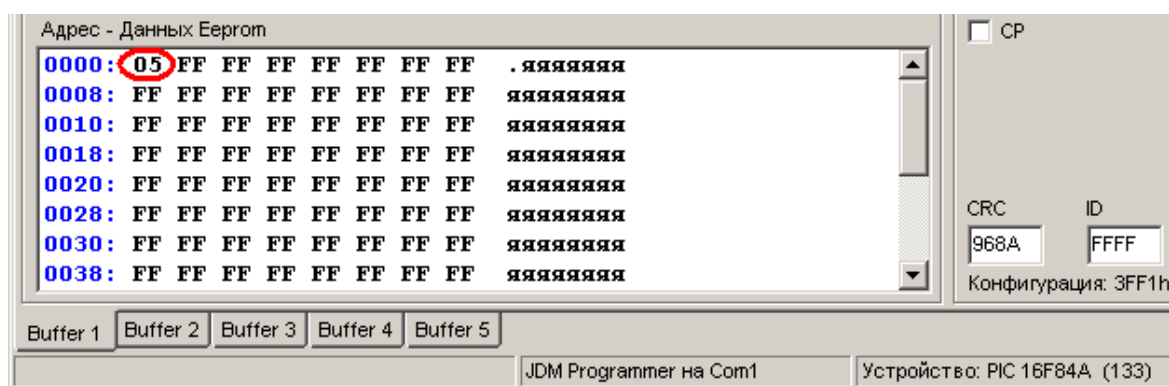
```

Несложно убедиться в незначительности сделанных модификаций текста программы. Тем не менее, это новый качественный подход в работе устройства. Прокомментируем работу нашей программы.

В шапке программы мы прописали регистры для работы с ПЗУ. Между шапкой и текстом программы стоит пара команд предварительной записи. Для чего она нужна? После прошивки и первого включения МК содержимое регистров ПЗУ нам неизвестно; также и МК туда еще не успел сделать запись числа (чисел). Для того чтобы точно знать содержимое в момент первого включения (не путать с повторным включением) мы путем предварительной записи помещаем в нулевой регистр ПЗУ число 5. Глядя на этот вопрос с другой стороны, мы можем в начале программы организовать проверку содержимого нулевого регистра ПЗУ и если оно не входит в диапазон 0...9 прописать туда из программы нужное число, например 5. Мы решили не усложнять алгоритм программы, т.к. после повторного включения в регистре ПЗУ будет находиться "правильное" число.

Итак, предварительная запись числа (чисел) в ячейку ПЗУ – это ни что иное, как процедура конкретного определения содержимого ячеек ПЗУ. Запись чисел в ПЗУ делается в момент прошивки МК из программы IC-Prog. Строчки предварительной записи не входят в текст программы, они указывают – что и куда записать. Тем не менее, данные, помещенные в ПЗУ в ходе предварительной записи, могут сразу использоваться для расчетов и только потом из программы изменяться.

После открытия файла прошивки (пункт 10) в окне данных EEPROM мы видим предварительно записанную нами пятерку в ОЗУ (в шестнадцатеричном формате):



Необходимо отметить, что прошиваемые данные EEPROM (данные ПЗУ) из оболочки IC-Prog могут быть изменены вручную. Попробуйте вставить любое число в нулевую ячейку EEPROM из диапазона 00...09, прошить МК и сделать первое включение. Если вы поняли зависимость ваших действий, следовательно, вы овладели новыми знаниями, а именно как без строчек предварительной записи заполнить ПЗУ из IC-Prog!

Программный код также можно менять из IC-Prog, но для этого необходимы знания машинного кода. Увы, у нас таких знаний нет.

Предлагаю вам выключить устройство в тот момент, когда на индикаторе у нас отображен символ девяти "9". А теперь МК вставим в программатор и считаем содержимое нашего МК из IC-Prog. Нажимаем кнопку



"Читать микросхему". Теперь смотрим в окно области данных EEPROM и обнаруживаем запись числа 09.

Все остальные действия в программе по работе с ПЗУ ранее прокомментированы.

Далее рассмотрим программу, которая после 5-го включения блокирует работу прибора (из примера 7) и выдает короткие звуковые сигналы.

	LIST	P=PIC16F84A
	__CONFIG	H3FF1
W	EQU	0
F	EQU	1
PC	EQU	H0002
STATUS	EQU	H0003
PORTA	EQU	H0005
PORTB	EQU	H0006
TRISA	EQU	H0005
TRISB	EQU	H0006

```

INTCON      EQU      H000B
C           EQU      0
Z           EQU      2
Reg_1       EQU      H000C
Reg_2       EQU      H000D
Reg_3       EQU      H000E
Reg_4       EQU      H000F      ; регистр под результат
; Определение регистров для работы с памятью (операции чтения/записи)
EEData      equ       08h      ; EEPROM - данные
EECon1      equ       08h      ; EECON1 - банк1.
EEAdr       equ       09h      ; EEPROM - адрес
EECon2      equ       09h      ; EECON2 - банк2.
            org       2100h      ; Обращение к EEPROM памяти данных.
            DE        5         ; предварительная запись
            org       0         ; начало программы

; подготовительные моменты
            bsf        STATUS,5   ; переход в Банк 1
            movlw     b00011111
            movwf     TRISA
            clrf      TRISB
            bcf       STATUS,5    ; переход назад в Банк 0

; чтение значения из памяти
            call       Read
; проверка числа кол-ва включений на равенство нулю
            bcf       STATUS,Z    ; опустим флаг Z в ноль
            movf      Reg_4,F     ; копировать из Reg_4 в Reg_4
            btfsc     STATUS,Z    ; делаем бит-проверку Z-флага
; если Z=1, то выполняется следующая инструкция, иначе - пропускается
            goto      m6
            goto      m7

; закидывание для случая ограничения работы (звуковые сигналы)
m6          clrf      PORTB
            call      Pause
            bsf       PORTB,7
            call      Pause
            goto      m6

; уменьшение на единицу кол-ва включений
m7          decf      Reg_4,F
            call      Write
; выход в режим нормальной работы
            movlw     b01101111
            movwf     PORTB
            clrf      Reg_4

; отслеживание нажатий кнопок
m3          btfss     PORTA,2     ; бит-проверка ножки RA2 - уменьшение
            goto      m1
            btfss     PORTA,3     ; бит-проверка ножки RA3 - увеличение
            goto      m2
            goto      m3         ; закидывание проверки

; проверка на ноль (на крайнее значение) и уменьшение значения регистра
m1          bcf       STATUS,Z    ; опустим флаг Z в ноль
            movf      Reg_4,F     ; копировать из Reg_4 в Reg_4
            btfsc     STATUS,Z    ; делаем бит-проверку Z-флага
; если Z=1, то выполняется следующая инструкция, иначе - пропускается
            goto      m4         ; переходим на отрисовку значения
            decf      Reg_4,F     ; уменьшить значение на 1 и сохранить
            goto      m4

; проверка на 9 (на др. крайнее значение) и увеличение значения регистра
m2          bcf       STATUS,C    ; опускаем флаг C в ноль
            movlw     .247        ; (255-9)+1 = 247 -> W
            addwf     Reg_4,W     ; (Reg_4)+W
            btfss     STATUS,C    ; делаем бит-проверку C-флага

```

```

; если бит C=0, то выполняется следующая инструкция
; если бит C=1, то следующая инструкция пропускается
        goto          m5
        goto          m4
m5      incf          Reg_4,F      ; увеличить значение на 1 и сохранить
m4      movwf         Reg_4,W
        call          TABLE
        movwf         PORTB
        call          Pause
        call          Write      ; вставка записи
        goto          m3

;=====
TABLE   addwfb        PC,F        ; Содержимое счетчика команд PC = PC + W
        retlw         b01101111 ; 0
        retlw         b00001100 ; 1
        retlw         b01011011 ; 2
        retlw         b01011110 ; 3
        retlw         b00111100 ; 4
        retlw         b01110110 ; 5
        retlw         b01110111 ; 6
        retlw         b01001100 ; 7
        retlw         b01111111 ; 8
        retlw         b01111110 ; 9

;=====
;delay = 500000 machine cycles
Pause   movlw         .85
        movwf         Reg_1
        movlw         .138
        movwf         Reg_2
        movlw         .3
        movwf         Reg_3
wr      decfsz        Reg_1, F
        goto          wr
        decfsz        Reg_2, F
        goto          wr
        decfsz        Reg_3, F
        goto          wr
        return

;=====
; запись данных в энергонезависимую память EEPROM (ПЗУ)
Write   clrf          INTCON      ; Глобальный запрет прерываний
        clrw          ; обнулить W, т.е. установить адрес "0"
        movwf         EEAdr      ; Скопировать W в регистр EEAdr
W       movf          Reg_4,W      ; Скопировать данные из Reg_4 в регистр
        movwf         EEData      ; Скопировать W в EEPROM
        bsf           STATUS,5    ; 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
        bcf           STATUS,5    ; Переход в нулевой банк.
        return

;=====
; чтение данных из энергонезависимой памяти EEPROM (ПЗУ)
Read    bcf           STATUS,5    ; Переход в нулевой банк.
        clrw          ; обнулить W, т.е. установить адрес "0"
        movwf         EEAdr      ; Скопировать W в регистр EEAdr
        bsf           STATUS,5    ; Переход в первый банк.

```

```

        bsf          EECon1,0      ; Инициализировать чтение.
        bcf          STATUS,5      ; Переход в нулевой банк.
        movf         EEData,W      ; Скопировать в W из EEPROM
        movwf        Reg_4         ; Скопировать из W в регистр Reg_4
        return
;=====
        end                      ; конец программы

```

Далее текст прошивки:

```

:0200000040000FA
:1000000083161F30850086018312542003118F0848
:1000100003190B28102886013820861738200B2852
:100020008F0345206F3086008F01051D1A28851D1E
:100030002028152803118F08031927288F03272844
:100040000310F7300F07031C262827288F0A0F08F4
:100050002D20860038204520152882076F340C3467
:100060005B345E343C34763477344C347F347E34C5
:1000700055308C008A308D0003308E008C0B3E286A
:100080008D0B3E288E0B3E2808008B010301890052
:100090000F0888008316081555308900AA3089009A
:1000A0008814081283120800831203018900831642
:0A00B0000814831208088F000800EE
:02400E00F13F80
:024200000500B7
:00000001FF

```

Выделенное жирным шрифтом в тексте программы представляет суть поставленной задачи. Специальные комментарии не требуются, т.к. основные пояснения сделаны в тексте программы. Для правильной работы устройства с алгоритмом блокировки требуется между включениями выдерживать паузу 2...5 сек для полной разрядки цепей питания.

Для самостоятельной работы предлагаю объединить в отдельной программе два свойства: сохранение значения и ограничение работоспособности. Разумеется, для этого вам потребуется два регистра ПЗУ и, как следствие, более интеллектуальный выбор адреса ПЗУ в сегментах чтения и записи. Не упускайте возможности провести гимнастику ума.

Глава 6. ИНСТРУМЕНТЫ MPLAB

В ходе написания программ нас не покидает естественный вопрос: насколько правильно будет работать МК по задуманному нами алгоритму. Хотим уверить вас – МК всегда работает правильно, неправильным может быть только алгоритм. В этой ситуации следует тщательно просмотреть программу, от строчки к строчке, рассмотреть все переходы и логику их выполнения, прочувствовать механизм работы всей программы в цикле. Это невозможно сделать без знания и понимания команд ассемблера.

Далее мы познакомимся с инструментами MPLAB, которые позволят визуально оценить последовательность выполнения команд, облегчат понимание работы МК и выполнения программы в целом. Предполагается рассмотреть следующие вопросы:

- симулятор MPLAB SIM;
- анализ регистров общего и специального назначения;
- измерение времени выполнения программы.

Обладая этими знаниями вы сможете сделать анализ правильности работы устройств на МК с использованием инструментов MPLAB.

Симулятор MPLAB SIM

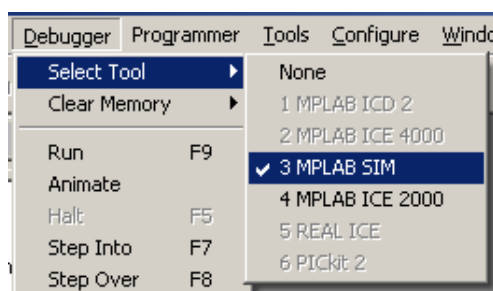
В данном разделе мы рассмотрим как в интегрированной среде проектирования MPLAB можно отслеживать ход выполнения программы.

Инструмент MPLAB SIM (далее симулятор) очень полезен при разработке и анализе выполнения программы. С помощью симулятора вы в реальном масштабе времени можете шаг за шагом просмотреть ход выполнения программы, убедиться в правильности логики и далее почувствовать реализацию машинной математики.

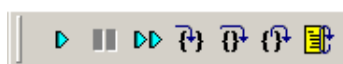
Будем считать, что в MPLAB создан проект, в котором открыта программа из любого примера, рассмотренных в главе 5. Проект должен быть успешно скомпилирован.

Для запуска симулятора нажимаем:

Debugger → Select Tools → MPLAB SIM.




Справа сверху появится панель следующего вида:




В нашем примере выполним следующую последовательность действий:

Нажимает кнопку Reset  – сброс к началу программы.

Появится зелёная стрелка  на полях, слева от строчек программы. Эта стрелка показывает нахождение рабочей точки программы.

Нажимаем кнопку Animate , что приведёт к непрерывному движению рабочей точки программы.

Нажимаем кнопку Halt , что приведёт к остановке движения рабочей точки программы.

Нажимаем кнопку Step Into , что приведёт к смещению рабочей точки на один шаг.

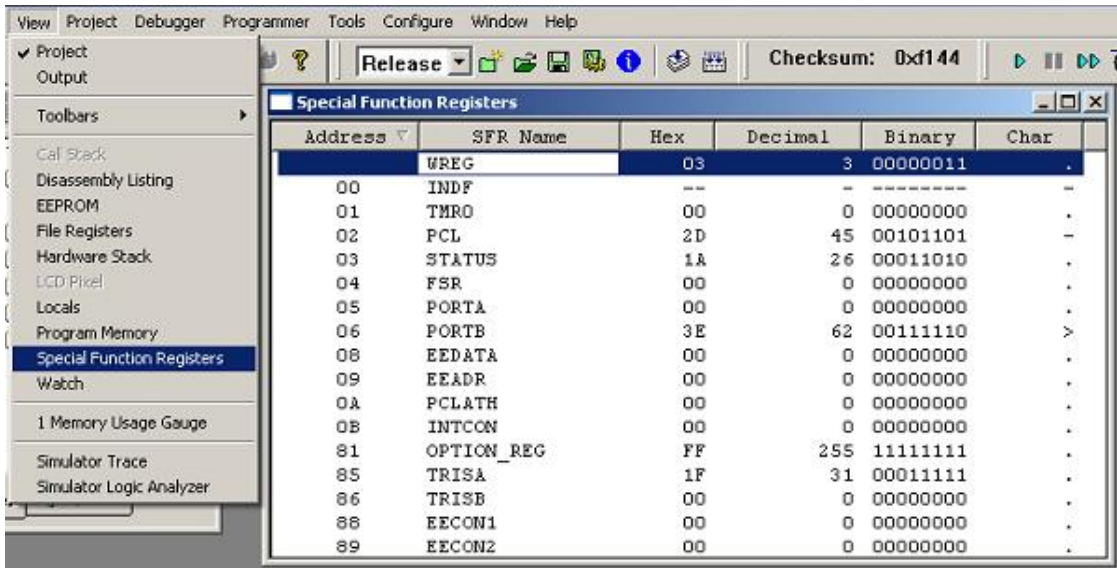
Таким образом, мы видим как в соответствии с текстом программы выполняются переходы между командами, от строчки к строчке.

Анализ регистров общего и специального назначения

С помощью симулятора MPLAB SIM мы научились смотреть ход выполнения программы. Однако, гораздо больший практический интерес представляет анализ содержимого регистров общего и специального назначения.

Начнем с регистров специального назначения. Откроем программу и включим симулятор (см. раздел "Симулятор MPLAB SIM"). Далее откроем окно регистров специального назначения.

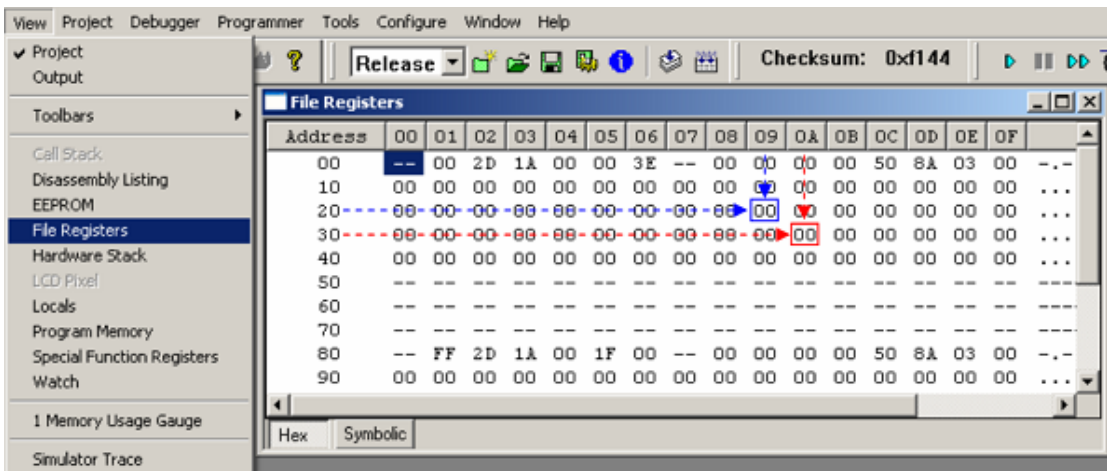
Нажимаем View → Special Function Registers:



В таблице окна "Special Function Registers" несложно проследить за изменением регистров специального назначения в ходе работы симулятора.

Не менее интересно отслеживать и состояние регистров общего назначения.

Нажимаем View → File Registers.



В ячейках таблицы окна "File Registers" мы видим числа в шестнадцатеричном формате. Это ни что иное как содержимое ВСЕХ ячеек оперативной памяти, а именно регистров общего и специального назначения. Таблица очень наглядна в плане охвата наблюдения за всеми регистрами, но не совсем понятна в плане понимания адреса регистра и его имени. Также определённые затруднения вызывает шестнадцатеричный формат данных в ячейках. Окно "Special Function Registers" в этом плане гораздо удобнее.

Адрес регистра (ячейки памяти, ячейки таблицы) в шестнадцатеричном формате несложно

вычислить путем сложения числа из первого столбца и числа из первой строки, на пересечении которых находится ячейка (см. глава 1, раздел "Регистры"). Например, $h20+h09 = h29$ или $h30+h0A = h3A$, что и является искомым адресом.

В окне "File Registers" удобно наблюдать за состоянием регистров при работе симулятора в режиме "Animate", особенно за регистрами, на которых реализованы задержки.

Необходимо отметить, что в рассматриваемых окнах можно вручную изменить состояние любого бита или любого регистра. Достаточно двойным щелчком мыши выделить значение и изменить содержимое ячейки. В нашем примере, изменяя величину значений регистров общего назначения (Reg_3 по адресу $h0E$ и др.) мы, соответственно, увеличиваем или уменьшаем время работы симулятора на локальном участке сегмента паузы. Изменить значение можно в любой момент работы симулятора. В отдельных случаях будет полезной возможность изменения битов в регистре TRISA, например при анализе работы программы с кнопками.

Измерение времени исполнения программы

В данном разделе мы рассмотрим как в MPLAB измерить время исполнения локального участка программы.

Нам известно, что МК никогда не прекращает выполнение программы. Согласно циклической концепции для правильной организации мы должны таким образом составить программу, чтобы она всегда выполнялась. Наша программа состоит из модулей (сегментов), выполняющие те или иные задачи. Иногда требуется знать, сколько времени выполняется тот или иной сегмент или интервал в сегменте. Эти знания нужны для оценки возможностей и характеристики устройства. В отдельных случаях без этих знаний невозможно синхронизировать работу с другими устройствами.

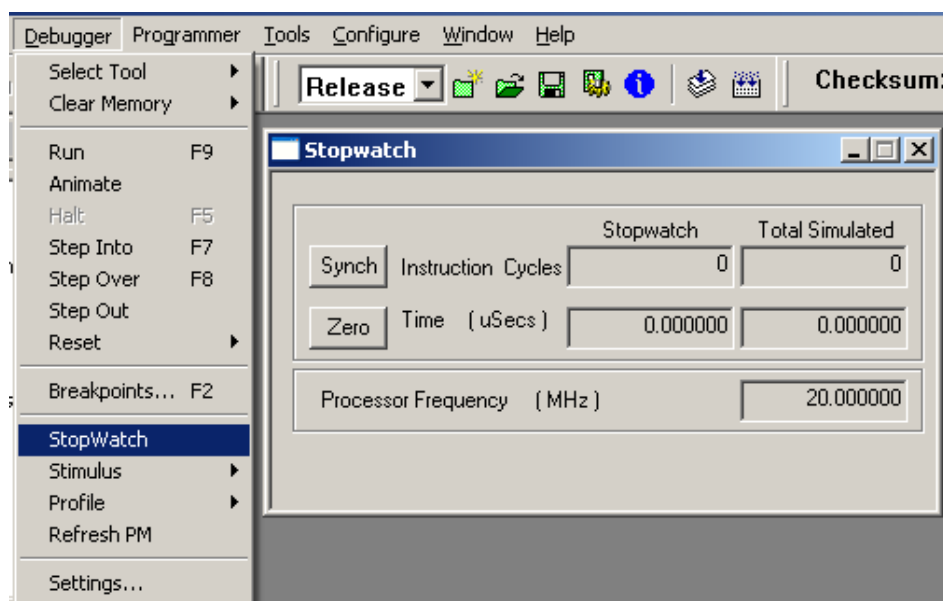
Для измерения мы будем использовать программу из примера 6 – "Работа нескольких кнопок. Многозадачность". Точнее мы будем измерять и вычислять период мигания символа нижнего подчеркивания.

Будем считать, что в MPLAB создан проект, в котором открыта программа из примера 6 (см. главу 5). Проект должен быть успешно скомпилирован. Требуется измерить время выполнения программы в сегменте отслеживания нажатия кнопок.

Последовательность действий:

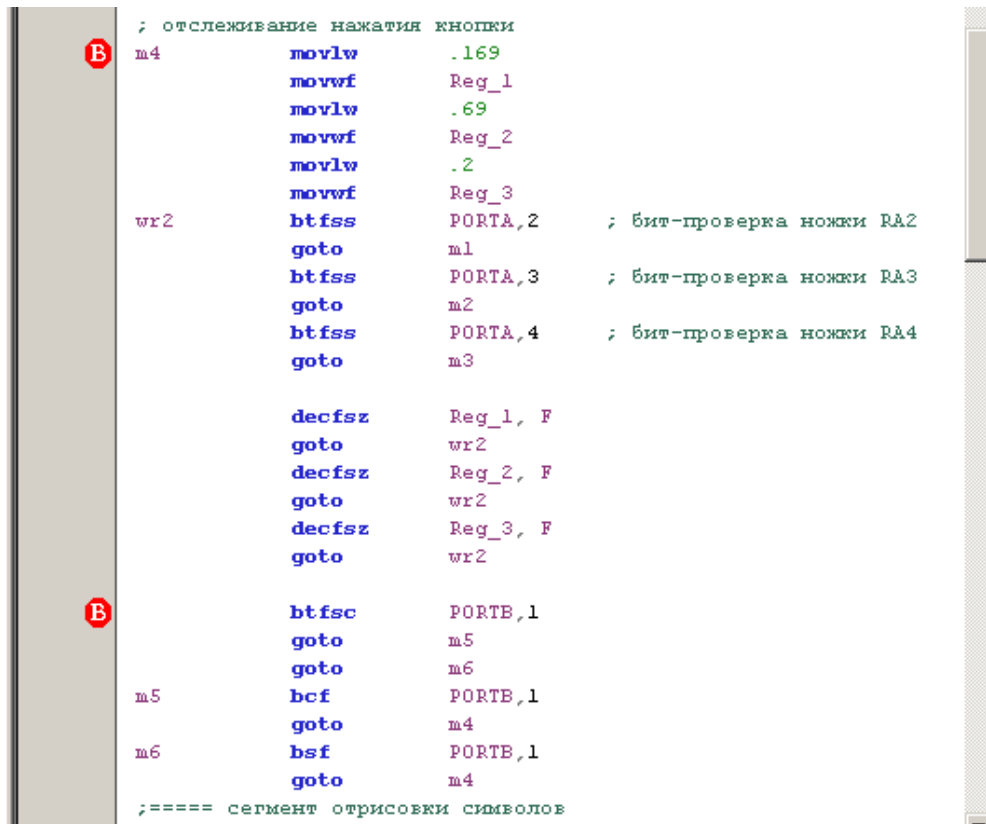
1. Включаем симулятор (см. раздел "Симулятор MPLAB SIM").
Нажимаем Debugger → Select Tools → MPLAB SIM.

2. Нажимаем Debugger → Stopwatch.



С помощью этого инструмента мы будем измерять задержку.

3. В тексте программы устанавливаем т.н. BreakPointy. Для этого двойным щелчком на полях напротив метки m4 и команды btfsc PORTB,1 устанавливаем маркеры. Слева должна появиться белая буква В в красном кружке. Мы будем измерять время задержки между этими маркерами. Повторный щелчок снимает маркер.



```

; отслеживание нажатия кнопки
m4      movlw    .169
        movwf    Reg_1
        movlw    .69
        movwf    Reg_2
        movlw    .2
        movwf    Reg_3
wr2     btfss    PORTA,2    ; бит-проверка ножки RA2
        goto     m1
        btfss    PORTA,3    ; бит-проверка ножки RA3
        goto     m2
        btfss    PORTA,4    ; бит-проверка ножки RA4
        goto     m3

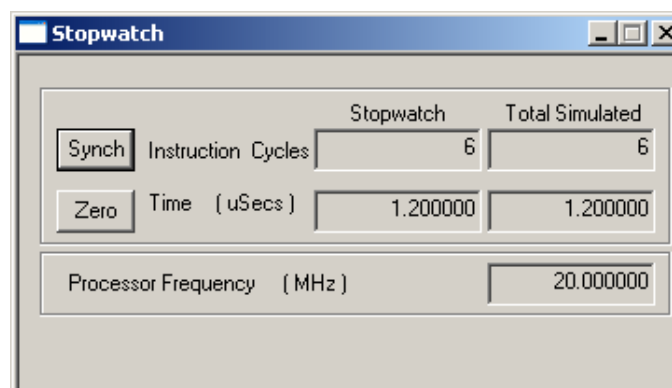
        decfsz   Reg_1, F
        goto     wr2
        decfsz   Reg_2, F
        goto     wr2
        decfsz   Reg_3, F
        goto     wr2

        btfsc    PORTB,1
        goto     m5
        goto     m6
m5      bcf       PORTB,1
        goto     m4
m6      bsf       PORTB,1
        goto     m4

;===== сегмент отрисовки символов

```

4. Нажимаем кнопку Play на панели симулятора, или щелкаем F9. В этом режиме будет происходить симуляция работы МК на компьютере, а остановится этот процесс тогда, когда мы доберемся до нашего первого BreakPointa. Смотрим на окошко Stopwatch. Его значения изменились:



Теперь нажимаем Zero. Этим мы сбрасываем наш секундомер в ноль. И снова жмем F9. Микроконтроллер начал работать дальше, и остановится он только когда доберется до второго BreakPoint'a. На этом интервале время выполнения программы занимает 250027 мкс или 0,25 сек.

Теперь несложно подсчитать период мигания символа нижнего подчеркивания $2 \cdot 0,25 = 0,5$ сек.

Глава 7. СОПРЯЖЕНИЕ МИКРОКОНТРОЛЛЕРА С ВНЕШНИМИ УСТРОЙСТВАМИ

В этой главе мы рассмотрим практические способы сопряжения МК с внешними устройствами. Под внешними устройствами мы понимаем любые элементы, которые могут управлять или могут быть управляемыми, либо с МК могут вести двухстороннее взаимодействие. Суть взаимодействия – обмен электрическими сигналами определенной длительности, определенной последовательности, на определенных линиях и, в отдельных случаях, определенной полярности. Иначе суть взаимодействия называется протоколом, где длительность определяет скорость обмена, последовательность состоит из набора сигналов высокого и низкого уровня, а определенная полярность характеризует аппаратную часть приемно-передающих устройств.

В главе предполагается рассмотреть следующие вопросы:

- описание интерфейса RS-232,
- передача данных в сторону компьютера,
- о кодовой таблице ANSI,
- электрическое сопряжение с ПК,
- работа с терминалом на ПК,
- приём данных от ПК на стороне МК,
- ... и хватит на этом, т.к. все остальные протоколы имеют схожую концепцию, а именно, передачу битов в байте определенными логическими уровнями определенной длительности.

Описание интерфейса RS-232

Интерфейс RS-232-C соединяет два устройства. Линия передачи первого устройства соединяется с линией приема второго и наоборот (полный дуплекс). Для управления соединенными устройствами используется программное подтверждение (введение в поток передаваемых данных соответствующих управляющих символов). Возможна организация аппаратного подтверждения путем организации дополнительных RS-232 линий для обеспечения функций определения статуса и управления.

Компьютер имеет 25-контактный (DB25P) или 9-контактный (DB9P) разъем для подключения RS-232C. Назначение контактов разъема приведено в таблице.

Наименование линий RS-232, их направление и номер контакта

(IN – в сторону ПК, OUT – со стороны от ПК)

Наименование	Направление	Описание	Контакт	Контакт
			(9-контактный разъем)	(25-контактный разъем)
DCD (RLSD)	IN	Carrie Detect (Определение несущей)	1	8
RxD	IN	Receive Data (Принимаемые данные)	2	3

TxD	OUT	Transmit Data (Передаваемые данные)	3	2
DTR	OUT	Data Terminal Ready (Готовность терминала)	4	20
GND	–	System Ground (Корпус системы)	5	7
DSR	IN	Data Set Ready (Готовность данных)	6	6
RTS	OUT	Request to Send (Запрос на отправку)	7	4
CTS	IN	Clear to Send (Готовность приема)	8	5
RI	IN	Ring Indicator (Индикатор)	9	22
FG	–	(Экран)	корпус разъема	корпус разъема

Назначение сигналов

DCD (RLSD) обнаружение несущей данных (детектирование принимаемого сигнала)

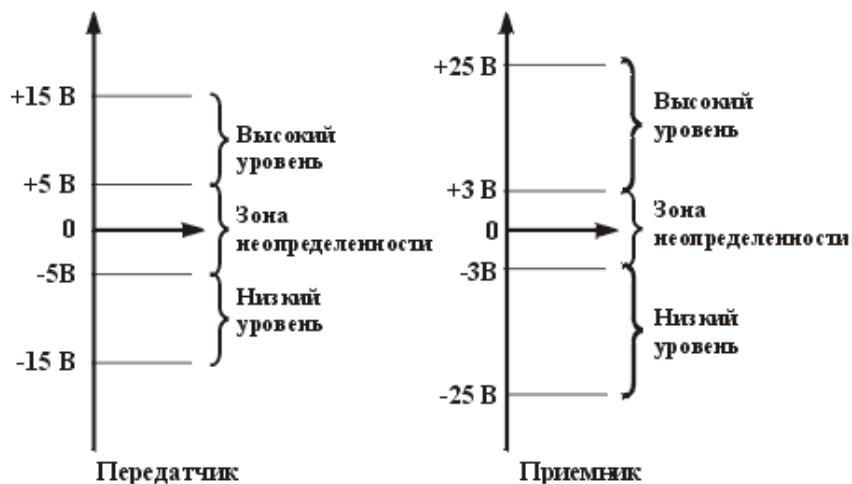
RxD	данные, принимаемые компьютером в последовательном коде (логика отрицательная)
TxD	данные, передаваемые компьютером в последовательном коде (логика отрицательная)
DTR	готовность выходных данных
GND	сигнальное заземление, нулевой провод
DSR	готовность данных, используется для задания режима модема
RTS	сигнал запроса передачи, активен во все время передачи
CTS	сигнал сброса (очистки) для передачи, активен во все время передачи, говорит о готовности приемника
RI	индикатор вызова, говорит о приеме модемом сигнала вызова по телефонной сети
FG	защитное заземление (экран)

Для того чтобы отвлечься от сложностей описания линий интерфейса RS-232, всё вышеперечисленное объясним на примере работы COM-порта:

- на линиях DTR и RTS компьютер может устанавливать высокий и низкий уровни сигналов;
- по линиям DCD, DSR и CTS компьютер может определять внешние сигналы соответствующих логических уровней;
- по линии RI компьютер может зафиксировать факт перевода по внешнему сигналу из логического низкого в логический высокий;
- по линиям TxD и RxD ведется обмен байтами, причем физическому низкому уровню соответствует логическая единица (т.е. передача инвертированный сигнал).

Как видим – всё достаточно просто. Назвать и описать линии можно как угодно. Нам важно понять направление работы линий и особенности линий для передачи данных. В отличие от параллельного порта, состоящего из восьми информационных линий и за один такт передающего байт, порт RS232, строго говоря, требует наличия только одной такой линии, по которой последовательно передается бит за битом. Это позволяет сократить количество информационных линий для передачи данных между устройствами, но уменьшает скорость.

Уровни сигналов RS-232C на передающем и принимающем концах линии связи



На данном рисунке под передатчиком понимается компьютер.

Все сигналы RS-232C передаются специально выбранными уровнями, обеспечивающими высокую помехоустойчивость связи.

Еще раз отметим, что данные передаются в инверсном коде (логической единице соответствует низкий уровень, логическому нулю – высокий уровень). Данные в один и тот же момент времени могут передаваться в обе стороны: по одной линии в одну сторону, по другой линии в другую сторону (дуплексный режим).



Последовательный поток данных состоит из битов синхронизации и собственно битов данных. Формат последовательных данных содержит четыре части: стартовый бит, биты данных (5-8 бит), проверочный и стоповый биты; вся эта конструкция иногда называется символом.

Когда данные не передаются, на линии устанавливается уровень логической единицы. Это называется режимом ожидания. Начало режима передачи данных характеризуется передачей уровня логического нуля длительностью в одну элементарную посылку. Такой бит называется стартовым. Получив стартовый бит, приемник выбирает из линии биты данных через определенные интервалы времени. Биты данных посылаются последовательно, причем младший бит первым, всего их может быть от пяти до восьми. За битами данных следует проверочный бит, предназначенный для обнаружения ошибок, которые возникают во время обмена данными. Последней передается стоповая посылка, информирующая об окончании символа. Стоповый бит передается уровнем логической единицы. Длительность стоповой посылки – 1, 1,5 или 2 бита. Специально разработанное электронное устройство, генерирующее и принимающее последовательные данные, называется универсальным асинхронным приемопередатчиком (Universal Asynchronous Receiver Transmitter, UART). В большинстве современных микроконтроллеров UART реализован на аппаратном уровне. В случае с PIC16F84A мы будем UART реализовывать программно.

Обмен информацией с помощью UART происходит следующим образом. Приемник обнаруживает первый фронт стартового бита и выжидает полтора тактовых интервалов, поскольку считывание должно начаться в середине первой посылки. Через один тактовый интервал считывается второй бит данных, причем это происходит в середине второй посылки. После окончания информационного обмена приемник считывает проверочный бит для обнаружения ошибок и стоповый бит, а затем переходит в режим ожидания следующей порции данных.

Скорость передачи информации в последовательном интерфейсе измеряется в бодах (бод – количество передаваемых битов за 1 сек). Стандартные скорости можно выбрать из ряда: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 бод (бит/с). Зная скорость в бодах, можно вычислить число передаваемых символов в секунду. Например, если имеется восемь бит данных без проверки на четность и один стоповый бит, то общая длина последовательности, включая стартовый бит, равна 10. Скорость передачи символов соответствует скорости в бодах, деленной на 10. Таким образом, при скорости 9600 бод будет передаваться 960 символов в секунду.

Проверочный бит предназначен для обнаружения ошибок в передаваемых битах данных. Когда он присутствует, осуществляется проверка на четность или нечетность. Если интерфейс настроен на проверку по четности, такой бит будет выставляться в единицу при нечетном количестве единиц в битах данных, и наоборот. Это простейший способ проверки на наличие одиночных ошибок в передаваемом блоке данных. Однако, если во время передачи искажению подверглись несколько битов, подобная ошибка не обнаруживается. Проверочный бит генерируется передающим UART таким образом, чтобы общее количество единиц было нечетным или четным числом в зависимости от настройки интерфейса; приемное устройство должно иметь такую же настройку. Приемный UART считает количество единиц принятых данных. Если данные не проходят проверку, генерируется сигнал ошибки.

Теперь рассчитаем длительность битовой посылки для всего ряда скоростей (1000000 микросекунд/скорость) и количество пересылаемых символов в секунду (скорость/10).

Скорость, бит/с	Длительность битовой посылки, микросекунд	Количество символов в сек.
110	9090,9	11
150	6666,6	15
300	3333,3	30
600	1666,7	60
1200	833,3	120
2400	416,7	240
4800	208,3	480
9600	104,2	960
19200	52,1	1920
38400	26,0	3840
57600	17,4	5760
115200	8,7	11520

Передача данных в сторону компьютера

В предыдущем разделе мы узнали протокол передачи интерфейса RS232 и длительность битовой посылки. Сейчас мы составим несколько элементарных программ для МК:

- передача от МК в сторону ПК набора символов;
- прием на стороне МК от ПК символов;
- двухсторонний обмен: ПК<=>МК.

Программа непрерывной передачи в сторону ПК фразы "Ура! ".
Периодичность посылки 1 сек.

```

LIST          P=PIC16F84A
__CONFIG     H3FF1

W            EQU            0
F            EQU            1
STATUS       EQU            H0003
PORTA        EQU            H0005
PORTB        EQU            H0006
TRISA        EQU            H0005
TRISB        EQU            H0006
C            EQU            0
Reg_1        EQU            H000C    ; для счетчика паузы 100 мкс
Reg_2        EQU            H000D    ; для счетчика битов в байте
Reg_3        EQU            H000E    ; для передаваемого символа
Reg_4        EQU            H000F
Reg_5        EQU            H0010
Reg_6        EQU            H0011

org          0                ; начало программы
; подготовительные моменты
Start        bsf             STATUS,5    ; переход в Банк 1
             movlw           b00011101 ; RA0 на выход, остальные на вход
             movwf           TRISA
             clrf            TRISB
             bcf             STATUS,5    ; переход назад в Банк 0
             bsf             PORTA,0     ; установка 1 - "режим ожидания"
             clrf            PORTB      ; очистка порта

```

```

; вставка символа "У" для передачи
m3      movlw      "У"          ; копируем букву У (рус) как символ,
который                                ; прописется в кодировке ANSI в Reg_4
(hC4)   movwf      Reg_3
        call       Tx
; о кодировке ANSI мы поговорим чуть позже
        movlw      "p"
        movwf      Reg_3
        call       Tx
        movlw      "a"
        movwf      Reg_3
        call       Tx
        movlw      "!"
        movwf      Reg_3
        call       Tx
        movlw      " "
        movwf      Reg_3
        call       Tx          ; т.о. отправлено "Ура! ",
        call       Pause2
        goto       m3          ; 5 символов, в т.ч. пробел
; сегмент передачи от МК в сторону ПК (9600, 8-N-1, 104 мкс)
Tx       movlw      .9          ; 8+1, т.е + бит С из STATUS
        movwf      Reg_2
        bcf        STATUS,C     ; подготовка стартового бита
m1       btfsf     STATUS,C
        goto       bit1
        goto       bit0
bit1     bsf        PORTA,1      ; передача единицы
        call       Pause
        goto       m2
bit0     bcf        PORTA,1      ; передача нуля
        call       Pause
        goto       m2
m2       rrf        Reg_3,F      ; сдвиг вправо для передачи с младшего
бита
        decfsz     Reg_2,F
        goto       m1
        bsf        PORTA,1      ; установка 1 - "режим ожидания"
        call       Pause
        return
;delay = 95 machine cycles
Pause    movlw      .31
        movwf      Reg_1
wr       decfsz     Reg_1, F
        goto       wr
        nop
        return

;delay = 1000000 machine cycles
Pause2   movlw      .173
        movwf      Reg_4
        movlw      .19
        movwf      Reg_5
        movlw      .6
        movwf      Reg_6
wr2      decfsz     Reg_4, F
        goto       wr2
        decfsz     Reg_5, F
        goto       wr2
        decfsz     Reg_6, F
        goto       wr2

```

```
return
```

```
end ; конец программы
```

Далее текст прошивки:

```
:020000040000FA
:1000000083161D3085008601831205148601D330C6
:100010008E001820F0308E001820E0308E0018205E
:1000200021308E00182020308E0018203020072824
:1000300009308D00031003181E28212885142A205A
:10004000242885102A2024288E0C8D0B1B2885142B
:100050002A2008001F308C008C0B2C280000080080
:10006000AD308F0013309000063091008F0B362892
:0A007000900B3628910B362808008B
:02400E00F13F80
:00000001FF
```

В случае передачи более длинных фраз (или набора слов), программу можно упростить, создав таблицу в которой символы из фразы будут следовать друг за другом, а обращение к таблице будет организовано из счетчика.

```
Start
; организация счетчика передаваемых символов
    movlw    .6          ; всего 5 символов +1
    movwf    Reg_4       ; 6 => Reg_4
; организация передачи из таблицы
m4    movf    Reg_4,W     ; Reg_4 => W
    call     TABLE      ; уход в таблицу
    movwf    Reg_3       ; символ из таблицы W => Reg_3
    call     Tx          ; уход в передачу
    decfsz   Reg_4,F     ; уменьшаем (Reg_4)-1
    goto     m4          ; зацикливание
    goto     Start
TABLE addwf    PC,F       ; Содержимое счетчика команд PC = PC + W
    nop      ; для корректировки
    retlw    " "
    retlw    "!"
    retlw    "A"
    retlw    "P"
    retlw    "y"
```

Размер таблицы накладывает некоторые ограничения, которые ранее мы не рассматривали для облегчения восприятия принципа работы. Выход на командные строчки таблицы производится путем принудительного увеличения счетчика команд PC. Этот счетчик команд организован с помощью обычного регистра, который может принять максимальное значение в 255. А теперь представим ситуацию, что перед таблицей находится некоторое количество других командных строк, например 32. Исходя из этого, таблица может содержать не более $(255 - 32) = 223$ строк, иначе будет переполнение регистра PC, что приведет к неправильному ходу выполнения программы. Это и есть ограничение размера таблицы для вышеуказанного примера.

Самостоятельно предлагаем сделать следующие модификации программы:

- посылка более длинных фраз (или набора слов) с использованием таблицы;
- посылка символов (фраз) по факту нажатия кнопки (кнопок);
- использование временных задержек между словами.

О кодовой таблице ANSI

Символы в тексте программы, заключенные в двойные кавычки " ", интегрированная среда проектирования MPLAB интерпретирует как символы из таблицы ANSI. Каждый символ в этой таблице кодируется 8-битным числом. Разновидность набора ANSI, содержащая символы русского

алфавита, называется Windows-1251.

Описание некоторых непечатных символов:

9 – Табуляция; 11 – Новая строка; 13 – Конец абзаца, 32 – Пробел.

Таблица соответствия чисел в разных системах счисления
и символов из таблицы ANSI (Windows-1251)

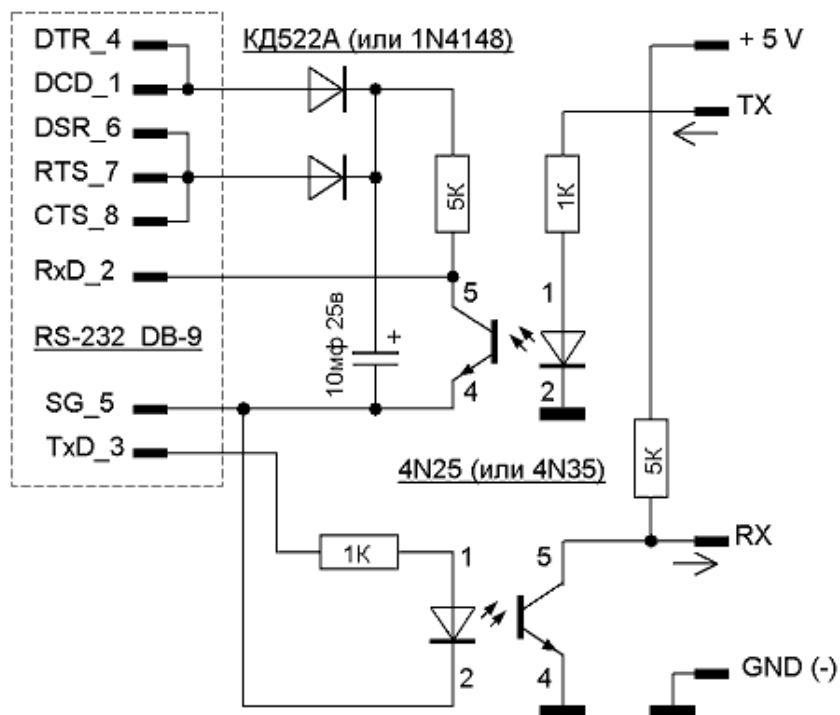
Д	В	Н	А	Д	В	Н	А	Д	В	Н	А	Д	В	Н	А
0	0000 0000	00		64	0100 0000	40	@	128	1000 0000	80	Ђ	192	1100 0000	C0	А
1	0000 0001	01		65	0100 0001	41	А	129	1000 0001	81	Ѓ	193	1100 0001	C1	Б
2	0000 0010	02		66	0100 0010	42	В	130	1000 0010	82	,	194	1100 0010	C2	В
3	0000 0011	03		67	0100 0011	43	С	131	1000 0011	83	ѓ	195	1100 0011	C3	Г
4	0000 0100	04		68	0100 0100	44	Д	132	1000 0100	84	„	196	1100 0100	C4	Д
5	0000 0101	05		69	0100 0101	45	Е	133	1000 0101	85	...	197	1100 0101	C5	Е
6	0000 0110	06		70	0100 0110	46	Ғ	134	1000 0110	86	†	198	1100 0110	C6	Ж
7	0000 0111	07		71	0100 0111	47	Г	135	1000 0111	87	‡	199	1100 0111	C7	З
8	0000 1000	08		72	0100 1000	48	Н	136	1000 1000	88	€	200	1100 1000	C8	И
9	0000 1001	09		73	0100 1001	49	И	137	1000 1001	89	‰	201	1100 1001	C9	Й
10	0000 1010	0A		74	0100 1010	4A	Ј	138	1000 1010	8A	Љ	202	1100 1010	CA	К
11	0000 1011	0B		75	0100 1011	4B	К	139	1000 1011	8B	‹	203	1100 1011	CB	Л
12	0000 1100	0C		76	0100 1100	4C	Л	140	1000 1100	8C	Њ	204	1100 1100	CC	М
13	0000 1101	0D		77	0100 1101	4D	М	141	1000 1101	8D	Ћ	205	1100 1101	CD	Н
14	0000 1110	0E		78	0100 1110	4E	Н	142	1000 1110	8E	Ќ	206	1100 1110	CE	О
15	0000 1111	0F		79	0100 1111	4F	О	143	1000 1111	8F	Џ	207	1100 1111	CF	П
16	0001 0000	10		80	0101 0000	50	Р	144	1001 0000	90	ђ	208	1101 0000	D0	Р
17	0001 0001	11		81	0101 0001	51	Q	145	1001 0001	91	‘	209	1101 0001	D1	С
18	0001 0010	12		82	0101 0010	52	Р	146	1001 0010	92	’	210	1101 0010	D2	Т
19	0001 0011	13		83	0101 0011	53	Ѕ	147	1001 0011	93	“	211	1101 0011	D3	У
20	0001 0100	14		84	0101 0100	54	Т	148	1001 0100	94	”	212	1101 0100	D4	Ф
21	0001 0101	15		85	0101 0101	55	У	149	1001 0101	95	•	213	1101 0101	D5	Х
22	0001 0110	16		86	0101 0110	56	В	150	1001 0110	96	–	214	1101 0110	D6	Ц
23	0001 0111	17		87	0101 0111	57	W	151	1001 0111	97	—	215	1101 0111	D7	Ч
24	0001 1000	18		88	0101 1000	58	X	152	1001 1000	98	~	216	1101 1000	D8	Ш
25	0001 1001	19		89	0101 1001	59	Y	153	1001 1001	99	™	217	1101 1001	D9	Щ
26	0001 1010	1A		90	0101 1010	5A	Z	154	1001 1010	9A	љ	218	1101 1010	DA	Ъ
27	0001 1011	1B		91	0101 1011	5B	[155	1001 1011	9B	›	219	1101 1011	DB	Ы
28	0001 1100	1C		92	0101 1100	5C		156	1001 1100	9C	њ	220	1101 1100	DC	Ь
29	0001 1101	1D		93	0101 1101	5D]	157	1001 1101	9D	ќ	221	1101 1101	DD	Э
30	0001 1110	1E		94	0101 1110	5E	^	158	1001 1110	9E	ћ	222	1101 1110	DE	Ю
31	0001 1111	1F		95	0101 1111	5F	_	159	1001 1111	9F	џ	223	1101 1111	DF	Я
32	0010 0000	20		96	0110 0000	60	`	160	1010 0000	A0		224	1110 0000	E0	а
33	0010 0001	21	!	97	0110 0001	61	a	161	1010 0001	A1	Ђ	225	1110 0001	E1	б
34	0010 0010	22	"	98	0110 0010	62	b	162	1010 0010	A2	Ѓ	226	1110 0010	E2	в
35	0010 0011	23	#	99	0110 0011	63	c	163	1010 0011	A3	Д	227	1110 0011	E3	г
36	0010 0100	24	\$	100	0110 0100	64	d	164	1010 0100	A4	„	228	1110 0100	E4	д
37	0010 0101	25	%	101	0110 0101	65	e	165	1010 0101	A5	Ѕ	229	1110 0101	E5	е
38	0010 0110	26	&	102	0110 0110	66	f	166	1010 0110	A6	ђ	230	1110 0110	E6	ж
39	0010 0111	27		103	0110 0111	67	g	167	1010 0111	A7	ѓ	231	1110 0111	E7	з

40	0010 1000	28	(104	0110 1000	68	h	168	1010 1000	A8	È	232	1110 1000	E8	и
41	0010 1001	29)	105	0110 1001	69	i	169	1010 1001	A9	©	233	1110 1001	E9	й
42	0010 1010	2A	*	106	0110 1010	6A	j	170	1010 1010	AA	€	234	1110 1010	EA	к
43	0010 1011	2B	+	107	0110 1011	6B	k	171	1010 1011	AB	«	235	1110 1011	EB	л
44	0010 1100	2C	,	108	0110 1100	6C	l	172	1010 1100	AC	¬	236	1110 1100	EC	м
45	0010 1101	2D	-	109	0110 1101	6D	m	173	1010 1101	AD		237	1110 1101	ED	н
46	0010 1110	2E	.	110	0110 1110	6E	n	174	1010 1110	AE	®	238	1110 1110	EE	о
47	0010 1111	2F	/	111	0110 1111	6F	o	175	1010 1111	AF	İ	239	1110 1111	EF	п
48	0011 0000	30	0	112	0111 0000	70	p	176	1011 0000	B0	°	240	1111 0000	F0	р
49	0011 0001	31	1	113	0111 0001	71	q	177	1011 0001	B1	±	241	1111 0001	F1	с
50	0011 0010	32	2	114	0111 0010	72	r	178	1011 0010	B2	İ	242	1111 0010	F2	т
51	0011 0011	33	3	115	0111 0011	73	s	179	1011 0011	B3	i	243	1111 0011	F3	у
52	0011 0100	34	4	116	0111 0100	74	t	180	1011 0100	B4	ı	244	1111 0100	F4	ф
53	0011 0101	35	5	117	0111 0101	75	u	181	1011 0101	B5	µ	245	1111 0101	F5	х
54	0011 0110	36	6	118	0111 0110	76	v	182	1011 0110	B6	¶	246	1111 0110	F6	ц
55	0011 0111	37	7	119	0111 0111	77	w	183	1011 0111	B7	·	247	1111 0111	F7	ч
56	0011 1000	38	8	120	0111 1000	78	x	184	1011 1000	B8	ë	248	1111 1000	F8	ш
57	0011 1001	39	9	121	0111 1001	79	y	185	1011 1001	B9	№	249	1111 1001	F9	щ
58	0011 1010	3A	:	122	0111 1010	7A	z	186	1011 1010	BA	€	250	1111 1010	FA	ъ
59	0011 1011	3B	;	123	0111 1011	7B	{	187	1011 1011	BB	»	251	1111 1011	FB	ы
60	0011 1100	3C	<	124	0111 1100	7C		188	1011 1100	BC	j	252	1111 1100	FC	ь
61	0011 1101	3D	=	125	0111 1101	7D	}	189	1011 1101	BD	S	253	1111 1101	FD	э
62	0011 1110	3E	>	126	0111 1110	7E	~	190	1011 1110	BE	s	254	1111 1110	FE	ю
63	0011 1111	3F	?	127	0111 1111	7F		191	1011 1111	BF	ï	255	1111 1111	FF	я

Эта таблица вам пригодится не один раз. Скачайте [этот файл с таблицей](#) и распечатайте.

Электрическое сопряжение с ПК

Далее рассмотрим схему электрического сопряжения нашей макетной платы с ПК для обеспечения соответствия уровней сигнала по спецификации RS-232. Мы говорили, что сигналы на линии должны отвечать определенному уровню и говорили, что данные передаются в инверсном коде. Для реализации поставленной задачи существует много схемотехнических решений, получивших название преобразователей уровней RS232/ТТЛ. Мы рассмотрим простейший вариант с использованием оптопар.



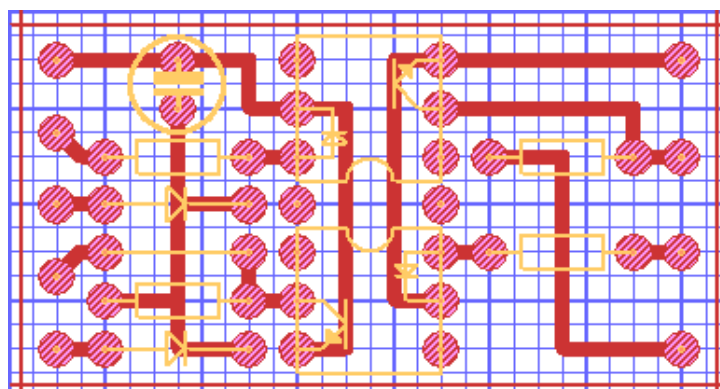
Использование оптопар обеспечивает гальваническую развязку между ПК и макетной платой. Питание преобразователя уровней со стороны ПК реализовано по линиям DTR и RTS. Факт подачи питания будет отслеживаться по линиям DCD (RLSD), DSR и CTS.

Для сборки преобразователя можно использовать любые кремниевые диоды подходящих габаритов, например, КД512. Оптопара 4N25 заменяема на 4N35 или на отечественную АОТ128. Конструктивно преобразователь соединяется с розеткой DB-9 через шнур из 5 линий. Для защиты оптопар от перегрева при пайке, установите на плату 14-контактную панель.

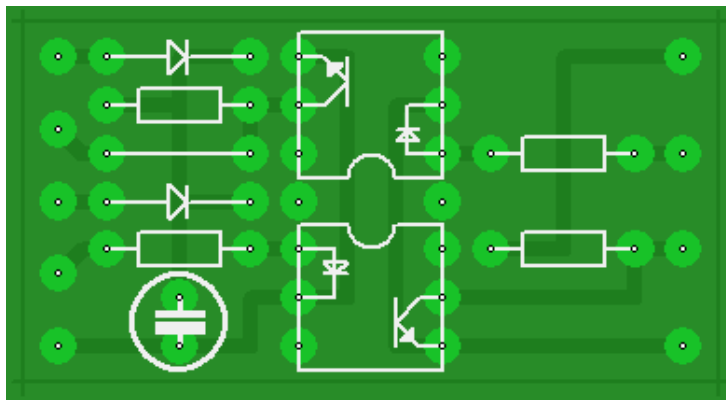
В этой и дальнейших схемах необходимо понимать способ связи между устройствами, а именно соединение линии приема одного устройства с передачей другого (и наоборот). В нашей программе для МК передача ведется с ножки RA1. В дальнейших программах мы организуем прием, который будет осуществляться на ножке RA0.

Ниже рисунок печатной платы преобразователя уровней.

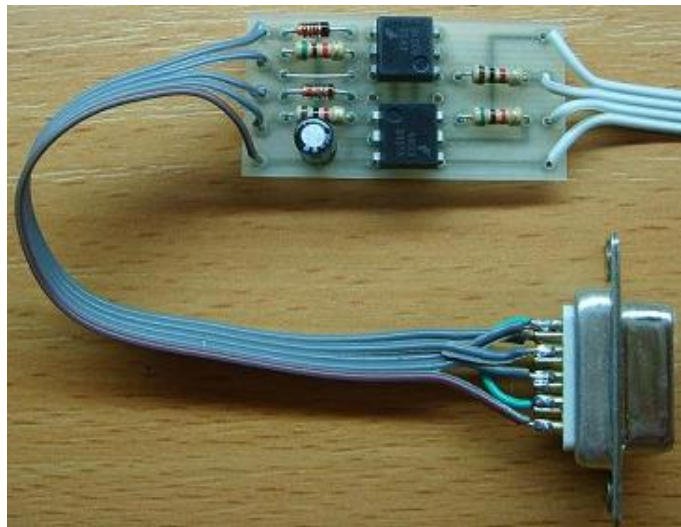
Вид со стороны печати



Вид со стороны элементов



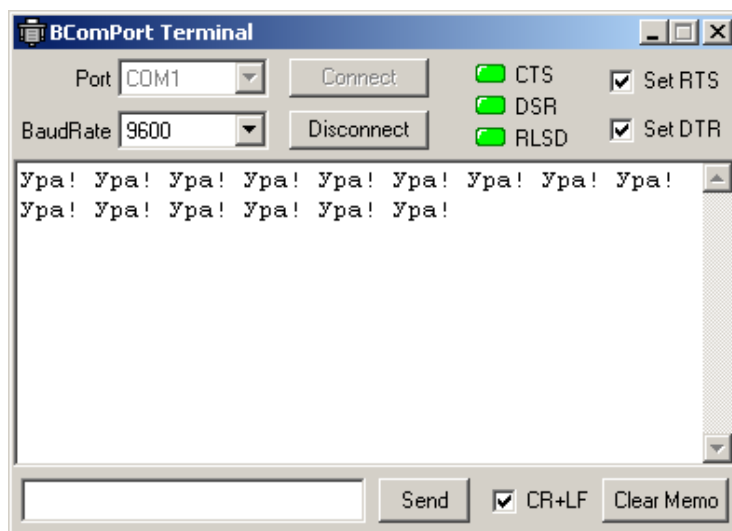
Фотография собранного преобразователя уровней



Работа с терминалом на ПК

Для проведения практического эксперимента обмена данными (точнее передачи данных в сторону ПК) нам потребуется соответствующая программа на ПК. Программу для приема и передачи данных через COM-порт иначе называют "Терминалом".

Можно привести массу готовых терминальных программ, однако, большую практическую пользу представляет очень простая программа "[ComPort Terminal](#)", которая работает под Windows 9X/ME/NT4/2K/XP.



Как проверить правильность работы программы и порта? Для этого на разъеме COM-порта соединяем (закорачиваем) линии TxD и RxD (выводы 3 и 2). Таким образом, соединив прием с передачей мы устанавливаем петлю на порту и можем сами себе отправлять и получать сообщения. Запускаем BComPort Terminal, отывается терминальное окно (рисунок выше). Выбираем порт и скорость, нажимаем кнопку Connect (т.е. открыть порт). Набираем в нижнем текстовом поле окна фразу и нажимаем кнопку Send. В большом текстовом поле должно появиться посланное сообщение.

К сведению. Если у вас есть два компьютера, то с помощью терминальной программы через COM-порт можно обмениваться сообщениями. Для этого потребуется три линии: общий провод GND (5), передача TxD (3) и прием RxD (2). Передачу на одном ПК соединяем с приемом на другом ПК и наоборот. Скорости передачи в запущенных программах, соответственно должны совпадать.

В окне терминала галочки Set RTS и Set DTR устанавливают высокий уровень на соответствующих линиях. Индикаторы линий CTS, DSR и RLSD показывают установившийся на них высокий уровень.

Будем считать, что у нас все приготовления успешно выполнены. Подключаем преобразователь уровней к макету (TX к RA1, RX к RA0). Соединяем преобразователь уровней с компьютером. Для удобства воспользуемся удлинительным шнуром от нашего программатора. Запускаем терминал, открываем порт (нажимаем Connect). Подаем питание на макет и устанавливаем факт передачи данных в сторону ПК. Ура!

Получив первый результат взаимодействия с ПК, не забудьте выполнить самостоятельную работу по модификации программы. Направления модификации обозначены ранее. Используйте эту возможность для оценки способностей делать программы под свои нужды.

(Дополнительно изучите [терминал для взрослых](#))

Приём данных от ПК на стороне МК

Настало время рассмотреть программу по приёму на стороне МК символов от компьютера. В нашем примере было принято решение не отягощать код программы сегментами по обработке принятого сигнала. Принятый 8-ми битный сигнал напрямую прописывается в порт В и, тем самым, на индикаторе (и звуковом излучателе) мы можем наблюдать комбинацию принятых битовых последовательностей. Последовательность (порядок) нулей и единиц в принятом 8-ми битном числе выясняем с помощью схемы макетной платы. Затем полученное число сверяем с таблицей ANSI.

```
LIST          P=PIC16F84A
__CONFIG      H3FF1
```

```
W            EQU            0
F            EQU            1
STATUS       EQU            H0003
```

```

PORTA      EQU      H0005
PORTB      EQU      H0006
TRISA      EQU      H0005
TRISB      EQU      H0006
C          EQU      0
Z          EQU      2
Reg_1      EQU      H000C      ; для счетчиков пауз
Reg_2      EQU      H000D      ; для счетчика битов в байте
Reg_3      EQU      H000E      ; для получаемого символа

          org      0          ; начало программы

; подготовительные моменты
Start      bsf      STATUS,5      ; переход в Банк 1
          movlw     b00011101 ; RA0 на выход, остальные на вход
          movwf     TRISA
          clrf      TRISB
          bcf      STATUS,5      ; переход назад в Банк 0
          clrf      PORTB        ; очистка порта

; прием и отрисовка принятого бита
m4         call     Rx          ; уход в приём
          movf      Reg_3,W
          movwf     PORTB        ; отрисовка полученного бита
          goto      m4

; сегмент приема данных от ПК на стороне МК (9600, 8-N-1, 104 мкс)
Rx         movlw     .8          ; ровно 8 бит
          movwf     Reg_2
m1         btfsc    PORTA,0      ; отслеживаем стартовый бит
          goto      m1
;delay = 52 machine cycles      ; задержка на 104/2 мкс
Pause1     movlw     .17         ; для выхода на середину бита
          movwf     Reg_1
wr1        decfsz   Reg_1, F
          goto      wr1
m3         call     Pause2
          btfsc    PORTA,0
          goto      bit1
          goto      bit0
bit1       bsf      STATUS,C
          goto      m2
bit0       bcf      STATUS,C
          goto      m2
m2         rrf      Reg_3,F      ; сдвиг вправо
          decfsz   Reg_2,F
          goto      m3
          call     Pause3
          return

;delay = 89 machine cycles      ; задержка с учётом
Pause2     movlw     .29         ; ранее выполненных команд
          movwf     Reg_1
wr2        decfsz   Reg_1, F
          goto      wr2
          nop
          return

;delay = 104 machine cycles      ; задержка под 1 стоповый бит
Pause3     movlw     .34
          movwf     Reg_1
wr3        decfsz   Reg_1, F
          goto      wr3
          nop
          return

          end                  ; конец программы

```

Далее текст прошивки:

```
:0200000040000FA  
:1000000083161D3085008601831286010A200E08A2  
:100010008600062808308D0005180C2811308C0049  
:100020008C0B10281F2005181628182803141A28CE  
:1000300003101A288E0C8D0B1228252008001D3065  
:100040008C008C0B21280000080022308C008C0BC7  
:0600500027280000080053  
:02400E00F13F80  
:00000001FF
```

По тексту программы сделаны комментарии. Зная, как организуется последовательный поток данных, несложно понять суть работы программы.

Для самостоятельного изучения рекомендуем вам сделать следующие модификации:

- отрисовывать на индикаторе определенные символы по факту послылки со стороны компьютера определенных букв или цифр;
- при послылке с компьютера чисел 1...8 на индикаторе макета переключать соответствующие сегменты (см. пример 5);
- создайте программу "кодового замка", согласно которой при послылке с компьютера последовательности символов (например, 721) срабатывал звуковой сигнал на макете (2 сек) и "замок" снова переходил в режим ожидания; если пришли три неправильные послылки, "замок" блокируется на 1 минуту.