В. Тимофеев
osa@pic24.ru

MPASM

Как правильно оформлять программы на ассемблере для PIC-контроллеров

(пособие для начинающих)

Содержание:

1 Вступление ....- 3 -
2 Структура текста программы....- 4 -
3 Файлы определений ....- 5 -
4 Определение переменных ....- 7 -
4.1 В абсолютном коде....- 7 -
4.2 В перемещаемом коде ....- 8 -
4.3 Два слова о директиве BANKSEL. ....- 9 -
5 Система именования идентификаторов ....- 10 -
5.1 Общие правила ....- 10 -
5.2 Правила именования переменных....- 10 -
5.3 Правила именования подпрограмм ....- 11 -
5.4 Правила именования констант ....- 12 -
5.5 Резюме ....- 12 -
6 Константы ....- 13 -
6.1 Неименованные константы ....- 13 -
6.2 Соблюдение системы счисления....- 13 -
6.3 Осмысленные значения констант ....- 14 -
7 Комментирование....- 16 -
7.1 Почему не пишут комментарии? ....- 16 -
7.2 Содержание комментариев....- 16 -
7.3 Что должно быть в комментариях....- 17 -
7.4 Чего в комментариях быть не должно ....- 17 -
7.5 Разделительный комментарий ....- 17 -
8 Опасный код ....- 18 -
8.1 goto $+n....- 18 -
8.2 Использование псевдокоманд ....- 20 -
8.3 Неполный перечень операндов ....- 21 -
9 О прерываниях ....- 23 -
9.1 Зачем нужен модуль прерываний....- 23 -
9.2 Характеристики обработчика прерывания ....- 23 -
9.3 Чего не должно быть в обработчике прерывания ....- 26 -
9.4 Как правильно запрещать/разрешать прерывания в программе....- 27 -
9.5 Доступ к переменным внутри прерывания ....- 28 -
10 Пример хорошо оформленной программы....- 32 -

1 Вступление

Существует распространенное заблуждение о том, что хорошее знание архитектуры контроллера – это гарантия качественного программирования. Это не так. Это все равно, что рассчитывать на то, что знание устройства стамески позволит вам заниматься резьбой по дереву.

Одной из характеристик качественного программирования является оформление исходных текстов программ. Многие недооценивают ее важность. Результатом пренебрежения качественным оформлением является не только не наглядный исходный текст программы, но и сложность сопровождения кода, отладки и повторного использования наработок. А внесение модификаций в такой код часто может привести к тому, что он перестает быть рабочим и начинает сбоить.

Возможно, многие уже сталкивались с последствиями плохого оформления исходников. Наверняка, заглядывая в программу спустя сравнительно большое время, чтобы внести незначительные исправления или чтобы освежить в памяти какие-то программные решения, многие сталкивались с тем, что разобраться в коде порой сложнее, чем написать все заново. Я уже не говорю о сложностях, возникающих при попытке разобраться с неаккуратным исходником, которым с вами кто-то поделился по доброте душевной (многие любят делиться своими наработками, не задумываясь об их качестве). Но самые большие неприятности возникают тогда, когда программа становится тяжело сопровождаемой, и, казалось бы, незначительные доработки (как, например, переназначение портов ввода вывода или пересчет временных задержек) требуют не только огромных трудозатрат, но и являются причинами внесения в программу трудноуловимых ошибок.

Откуда берется такая недооценка?

Первая причина – «так научили»: слышали такие объяснения от преподавателя в институте, заглядывали через плечо к знакомому программисту, читали статьи, книги или Интернет-ресурсы по программированию МК. Учитель является авторитетом, и все, что он делает и как он делает, воспринимается как истина.

Вторая причина – это лень и неряшливость. Многие часто, даже зная, что делают неаккуратно, оправдывают себя тем, что «это временно, а потом перепишу аккуратнее». Практика показывает, что это самое «потом» никогда не наступает. Поэтому сразу нужно делать хорошо. Да, это отнимает больше времени; да, в отличие от способа «тяп-ляп» или написания программы в лоб, аккуратность требует подготовки и предварительной проработки. Но гарантирую, что пройдет год, или два, или десять – и придет понимание, что все это было не зря, а вместе с пониманием – гордость за свою работу.

Есть еще **третья причина**, которая присуща начинающим программистам, сильно переоценивающим свои интеллектуальные способности. Они не пишут комментарии, считая, что и так все запомнят; они не утруждают себя проработкой имен идентификаторов, т.к. им и так понятно, что temp1 – это счетчик, temp2 – это таймер, а temp3 – это подпрограмма перевода двоичного числа в десятичное. Они делают еще много чего самонадеянного, что в конченом счете приводит к невозможности сопровождать собственноручно написанный код. **Рекомендую всем: пока нет многолетнего опыта, подтверждающего ваши незаурядные интеллектуальные способности и феноменальную память, не пренебрегайте аккуратностью.**

Для кого эта статья. Статья адресована программистам PIC-контроллеров младшего и среднего семейства, пишущим на языке ассемблера. Предполагается, что читатель знаком с архитектурой, набором инструкций и набором регистров специального назначения данных контроллеров.

Подход, изложенный мной в этой статье, - не единственно правильный. И, само собой, как и любой другой подход, годится не на все случаи жизни. Без труда можно придумать пример задачи, для решения которой данный подход окажется неэффективным. Но и в этом случае изложенным здесь материалом можно будет воспользоваться как шаблоном для формулирования собственных правил. Программа, ее стиль – это лицо программиста. По ее внешнему виду можно многое сказать о нем. Так что не теряйте лица! **J**

2 Структура текста программы

Как ни странно, многие даже не задумываются о такой важной характеристике исходного текста программы, как структурность. Когда вы берете в руки книгу, вы знаете, что у нее есть оглавление (или содержание), титульные и выпускные данные, список литературы, часто еще алфавитный указатель. Вы знаете, что элементы каждой из перечисленных групп находятся в одном месте, а не разбросаны по всей книге. Например, алфавитный указатель находится в самом конце, после содержательной части и перед оглавлением, что позволяет без труда найти нужное слово, словосочетание или термин и сразу открыть интересующую страницу.

Текст программы так же должен быть разбит на секции с тем, чтобы его было легко читать, сопровождать, документировать и модифицировать. Применительно к PIC-микроконтроллерам структура текста программы может выглядеть так:

- ***блок определений***
    - **секция заголовка** (с информацией о назначении программы, имени автора, дате создания и внесенных изменений, применяемом контроллере, тактовой частоте и т.д.)
    - **секция подключаемых файлов**
    - **секция конфигурации** (содержит определение битов конфигурации и IDLOC)
    - **секция определения констант**
    - **секция определения EEPROM-данных**
    - **секция определения макросов**
    - **секция объявления переменных**
- ***блок кода***
    - **вектор сброса** (обычно содержит только инструкцию безусловного перехода на код инициализации)
    - **обработчик прерывания**
    - **код инициализации**
    - **основной цикл программы**
    - **подпрограммы**
    - **END**

Для оформления секций внутри одного файла есть свои правила:

- Каждая секция должна содержать только те описания, которые ей соответствуют (т.е. не нужно в секции кода определять константы)
- Секции должны быть едиными, а не разделенными на несколько кусков, разбросанных по всему тексту программы
- Каждой секции должен предшествовать хорошо заметный блок комментария, содержащий название секции

См. параграф **“10. Пример хорошо оформленной программы”** для пояснений.

3 Файлы определений

Часто встречаются программы, которые авторы пытаются сделать самодостаточными настолько, что в самом тексте программы вручную определяют все регистры для данного контроллера. Это хорошо только при объяснении абсолютному новичку связи между текстом программы и адресами регистра конкретного контроллера. Но писать программы в таком стиле – это дурной тон, приводящий к существенному усложнению поддержки и модификации программы. Основной причиной некорректности такого подхода является то, что различные микроконтроллеры фирмы Microchip имеют регистры с одними и теми же именами, но расположенными по разным адресам. Типичный пример – расположение регистров управления EEPROM для PIC16F628 (адреса 9Ah-9Dh) и для, например, PIC16F819 (10Ch-10Fh, 18Ch-18Dh). При замене контроллера (неважно, по каким причинам) можно случайно пропустить такие различия, что приведет к неправильному поведению программы. А номенклатура 16x и 18x PIC'ов исчисляется десятками, и для адаптации исходника под каждый конкретный чип потребуется довольно много телодвижений.

Кроме того, при внесении списка регистров в файл вручную можно допустить ошибку, которая будет заметна не сразу (скажем, ошибку в определении INTCON будет заметно при первом же запуске программы, но, допустив ошибку при определении битов регистра PCON, мы можем лишить контроллер возможности восстановить свою работу после сбоя по Brown-Out-Detect; а операция это не частая и заметить эту ошибку в работающем устройстве не так просто).

Поэтому не стоит пренебрегать файлами определений, поставляемых в комплекте с MPASM. Они не только избавят от рутины переопределения всех регистров и их служебных битов вручную, но и позволят легко и быстро перенастроить исходный текст под другой контроллер.

Неправильно:

```
INDF    equ     00h
TMR0    equ     01h
PCL     equ     02h
status  equ     03h
…
```

Правильно:

```
#include <p16f628a.inc>
```

Также часто встречается ошибка, когда слово конфигурации задается в виде шестнадцатеричного представления, вместо использования предопределенных имен битов конфигурации. Так не только теряется наглядность, но и закладывается мина на будущее самому себе, т.к. в огромной массе микроконтроллеров линейки PIC16 встречается и такое, что даже у родственных МК биты конфигурации задаются по-разному. Типичный пример – PIC16F628 и PIC16F628A. Сравните формат слов конфигурации:

PIC16F628:

CP1	CP0	CP1	CP0	-	CPD	LVP	BODEN	MCLRE	FOSC2	PWRTE	WDTE	F0SC1	F0SC0

bit13 bit0

PIC16F628A:

CP	—	—	—	—	CPD	LVP	BOREN	MCLRE	FOSC2	PWRTE	WDTE	F0SC1	F0SC0

bit 13 bit 0

и обратите внимание на биты CP. Задание конфигурации для PIC16F628A в виде:

Неправильно:

```
__CONFIG 2130h
```

вызовет коллизии при сборке для PIC16F628. Но если эта коллизия еще разрешится на уровне оболочки MPLAB IDE (без оболочки сам ассемблер промолчит), которая сообщит нам, что неправильно заданы биты конфигурации (обе пары CP1:CP0 должны совпадать), то при применении контроллера PIC16F819 уже получим неприятности:

PIC16F819:

CP	CCPMX	DEBUG	WRT1	WRT0	CPD	LVP	BOREN	MCLRE	FOSC2	$\overline{PWRTEN}$	WDTEN	F0SC1	F0SC0

bit13 bit0

Ни ассемблер, ни интегрированная среда не выдадут предупреждения. Обратите внимание, что бит DEBUG окажется установленным в «0», т.е. внутрисхемный отладчик будет включен, и программа в автономном режиме просто не заработает. А разбирать шестнадцатеричное представление битов конфигурации и соотносить его с таблицей их значений из документации для каждого контроллера – это лишний труд, выполняя который, довольно легко ошибиться.

Правильно:

```
#include <p16f628a.inc>

__CONFIG _CP_OFF & _DATA_CP_OFF & _LVP_OFF & _BODEN_OFF & _MCLRE_ON & _HS_OSC & _WDT_ON
```

4 Определение переменных

4.1 В абсолютном коде

Учитывая архитектуру микроконтроллеров PIC10/12/16/18 и специфику инструкций, присущих данным семействам, появляется довольно много возможностей назначить имя ячейке памяти из области регистров общего назначения (т.е. объявить переменную). Как бы мы это не сделали:

```
      Counter equ    20h
      array    equ    22h
;-------------------------------------------------------

              ORG    0020h
      counter
              ORG    0022h
      array
;-------------------------------------------------------

      define  counter 20h
      define  array   22h
;-------------------------------------------------------

              CBLOCK   20h
              counter : 2
              array   : 8
              ENDC
```

мы сможем обращаться к ячейкам 20h и 22h регистровой памяти через их имена **counter** и **array**:

```
      decf    counter, f
      movlw   0x55
      movwf   array
```

При любом варианте определения (из приведенных выше) эту программу ассемблер переведет в один и тот же код. *(Примечание: существуют еще способы, например, директивами **constant** или **set**, но они мало отличаются от определения с помощью **equ**.)*

Получается, переменные можно объявлять как угодно, хоть через директиву задания смещения программного кода ORG? Да, можно. Но не нужно, и тому есть несколько причин.

Причина 1. при работе с чужим кодом, например, можно столкнуться с трудностями оценки используемых ресурсов. Предположим, что в начале программы встретилось:

```
      counter equ    20h
      array   equ    22h

      number  equ    32h
```

Как это понимать? **number** – это переменная? Ячейка 32h занята или свободна? А можно ли объявить переменную с адресом 25h? **array** – это массив какой размерности? И т.д. Придется искать идентификатор **number** по всему тексту программы и выяснять, как он применяется и является ли он адресом регистра, или же это просто константа для вычислений.

Причина 2. Хоть каждый идентификатор и имеет конкретное назначение, но не в каждом фрагменте кода это назначение можно однозначно понять:

```
      movlw   number
      call    PreparArray
```

Здесь однозначно не скажешь, передается ли в функцию число 32h, или туда передается адрес переменной для работы с ней через косвенную адресацию. Значит, нужно идти смотреть тело функции **PrepareArray**.

Причина 3. Когда вы садитесь обедать, для каждого блюда есть свой столовый прибор: суп вы едите с помощью столовой ложкой, макароны с котлетами – с помощью вилки, размешиваете сахар в чае чайной ложкой, хлеб режете ножом. Можно делать и все наоборот: суп хлебать чайной ложечкой, а сахар размешивать вилкой. Но это можно делать дома, когда вас никто не видит, а в ресторане вы просто испортите аппетит окружающим.

Это я к тому, что для каждого типа идентификатора есть свои средства описания. В частности, для определения переменных в регистровой памяти в ассемблере должна быть использована директива **CBLOCK**. Все остальные методы будут вводить в заблуждения и самого автора кода, и, уж тем более, тех, с кем он поделится своим творением.

Правильно:

```
CBLOCK   20h
counter :2
array   :8
ENDC
```

Добавлю еще, что, в отличие от трех других записей, в этой вы наглядно обозначаете, что **counter** – двухбайтовая переменная, а **array** – массив из 8 байт. И не нужно будет ползать по всему исходнику в поисках фрагментов кода с ее обработкой, чтобы узнать, можно ли в первом варианте записи воткнуть еще одну переменную с адресом 21h или 25h.

4.2 В перемещаемом коде

Если вы занимаетесь модульным программированием, то про абсолютные адреса лучше забыть. Работу по распределению адресов должен взять на себя линкер. В случае с перемещаемым кодом вариантов выделять память под переменные, к счастью, не так много. Выделение памяти под переменную производится с помощью директивы res, размещенной в соответствующей секции. В зависимости от времени жизни переменной (и от типа доступа к ней), для ее объявления можно использовать секции:

udata – для статических переменных, которые сохраняют за собой ячейку на протяжении всего времени работы программы;

udata_acs – для PIC18 – размещение переменных в ACCESS области (первые 128 или 96 байт ОЗУ);

udata_ovr – для оверлейных переменных, время жизни которых ограничено временем выполнения использующей их функции. Переменные, объявленные в этой секции могут перекрываться другими переменными даже из других модулей.

idata – для инициализируемых переменных.

Пример:

```
        udata
counter res     2
array   res     8
```

Ассемблер оттранслирует это в объектный код, в котором не будет конкретной привязки к адресам. Адреса будут назначены линкером при окончательной сборке проекта. Следует отметить, что на этапе

компиляции неизвестно, в каком банке окажутся эти переменные после сборки (однако, все переменные, объявленные в одной секции, окажутся в одном банке памяти), поэтому перед обращением к таким переменным следует пользоваться макросами banksel:

```
movlw    10h
banksel  counter
movwf    counter
```

4.3 Два слова о директиве BANKSEL.

Использование этой директивы всегда предпочтительнее, чем ручная установка битов **RP0**, **PR1**. Дело в том, что, во-первых, эта директива никогда не ошибется с выбором банка, а во-вторых, она позволяет делать код гибким так, что его можно будет успешно применять в контроллерах с различным количеством банков памяти, а также позволяет помещать переменные в любой банк без модификации кода. Также не лишним будет упомянуть, что битов **RP0** и **RP1** нет в контроллерах PIC18, PIC12F1xxx и PIC16F1xxx, поэтому применение **BANKSEL** позволит сделать код переносимым.

Примечание. *То же самое касается директивы **PAGESEL**, которую следует использовать вместо ручной установки битов 3 и 4 в регистре **PCLATH**.*

5 Система именования идентификаторов

Еще одна вечная проблема программистов – это каша в именовании идентификаторов. Пока программа небольшая, все выглядит вполне пристойно: есть переменные counter, data, timer и т.д. Но потом программа начинает разрастаться, появляется необходимость иметь еще один счетчик (многие, особо не задумываясь, просто называют его counter2), еще один таймер (называют timer2), еще один массив данных (называют data2) и пр. Потихоньку программа разрастается, и начинает появляться путаница в именах и назначениях идентификаторов, причем не только переменных, но и меток в программе.

Для ассемблера, да еще для абсолютного кода (когда линкер не участвует в генерации HEX-файла) довольно трудно сформулировать четкие правила именования идентификаторов. Основная сложность заключается в том, что область видимости всех идентификаторов – это вся программа. Таким образом, мы вынуждены добиваться того, чтобы все идентификаторы без исключения были уникальными (в модульном программировании все намного проще, т.к. область видимости локальных идентификаторов ограничивается модулем).

Особые сложности вызывает именование внутренних меток подпрограмм, т.к. многие подпрограммы имеют внутри себя циклы (которые всегда хочется назвать Loop), точки выхода (Exit), точки ошибок и т.д.

5.1 Общие правила

Тем не менее, примерные правила именования, которые сделают исходник более наглядным (а, следовательно, проще сопровождаемым), сформулировать можно. Ниже я попробую сформулировать правила для именования различных объектов программы с тем, чтобы, даже встречая в коде фрагмент, по действиям которого нельзя однозначно определить назначение идентификатора (вспомним пример с передачей в функцию числа Number), по типу записи идентификатора можно было бы догадаться о его назначении. Единых правил, особенно применительно к ассемблеру, нет, у каждого программиста они могут быть свои, но все они должны совпадать по четырем основным критериям:

- Имя должно быть осмысленным (т.е. не i, а counter)
- Имя должно быть содержательным (т.е. не counter, а bits_counter), т.е. отражать его назначение
- Имя не должно быть перегружено лишней информацией
- Если используется модульное программирование или подпрограмма имеет четкую функциональную привязку, то имя должно содержать префикс в виде имени модуля (или аббревиатуры модуля), отделенный от остальной части имени символом подчеркивания

5.2 Правила именования переменных

- Слова пишутся строчными буквами
- Слова разделяются символами подчеркивания

Примеры удачных имен:

```
i2c_bits_counter
lcd_data
rs232_byte_in
rS232_byte_out
relay_timer
```

Взглянув на имя любой переменной, можно сразу понять ее назначение.

Неудачные имена:

```
CounterOfI2CBits          ; Много лишнего
I                         ; неинформативно
ttt
Temp
temp2
Temp3
```

5.3 Правила именования подпрограмм

- Структура имени: <модуль>_<действие><объект>[<суффикс>]
  - Модуль – имя (или аббревиатура) модуля, если используется
  - Действие – глагол, определяющий назначение подпрограммы (Read, Count и т.п.)
  - Объект – существительное, определяющее параметрическую составляющую подпрограммы (Byte, Array, Checksum и т.п.)
  - Суффикс – необязательное поле, отражающую какую-либо дополнительную характеристику подпрограммы (ON, OFF, Rom и т.п.)
- Слова начинаются с заглавной буквы
- Слова пишутся без разделителей (не считая знак подчеркивания, отделяющий имя модуля)

Примеры удачных имен:

```
I2C_Readarray
CalcChecksum
TurnRelayON
TurnRelayOFF
```

Стоит добавить, что в некоторых случаях можно применять однословные названия, которые однозначно отражают назначение функции:

```
Delay                     ; Задержка
Sqrt                      ; Вычисление квадратного корня
```

Неудачные имена:

```
ReadDataFrom24LC64        ; Много лишнего
Relay                     ; Не содержательно
```

Как уже было сказано, особняком стоит проблема именования меток внутри подпрограмм, особенно для абсолютно размещаемого кода, где нет возможности ограничить область видимости имен идентификаторов. На сегодняшний день сколько-нибудь эффективного и красивого метода назначения имен таким меткам я не знаю. Собственно, от написания программ на ассемблере я давно отошел, но когда им занимался, имена назначал следующим образом: давал каждому имени префикс в виде аббревиатуры подпрограммы, в которой находится метка, а дальше добавлял смысловую часть, например:

```
I2C_SendByte:

      movlw   8
      banksel i2c_bits_counter
      movwf   i2c_bits_counter

I2CSB_Loop:                        ; Префикс в виде аббревиатуры I2CSB_
      ...
      decfsz  i2c_bits_counter, f
      goto    I2CSB_Loop

      return
```

Данное решение мне видится не очень удачным, однако, я его долго использовал, т.к. оно практически исключало коллизии при стыковке большого количества подпрограмм в одном проекте.

5.4 Правила именования констант

- Именуются только заглавными буквами
- Слова разделены символом подчеркивания
- Константам, обозначающим порты ввода вывода, следует давать префикс PIN_ или PORT_

Примеры удачных имен:

```
      USART_SPEED
      I2C_ADDR_WIDTH
      TIMER1_PERIOD
      TRISB_CONST
      PIN_GREEN_LED
      PIN_BUTTON
```

5.5 Резюме

Я привел пример системы именований, которая позволяет задавать однозначно интерпретируемые имена идентификаторов. Естественно не следует забывать, что в коде имена идентификаторов следует писать по тем же правилам, как они записаны при определении. Если в ассемблере отключена чувствительность к регистру, то, конечно, можно переменную определить как lcd_data, а обращаться к ней через идентификатор LCD_DATA. С точки зрения ассемблера ошибки не будет, но концептуальная ошибка налицо.

Вспомним наш пример:

```
      movlw   number
      call    PrepareArray
```

По такой записи, учитывая эту систему именования, однозначно можно сказать, что number – это переменная, а в функцию передается ее адрес. Если бы была такая запись:

```
      movlw   NUMBER
      call    PrepareArray
```

то было бы понятно, что в функцию передается константа.

6 Константы

6.1 Неименованные константы

Дурным тоном в программировании считается использование неименованных констант в оперативной части программы (т.е. в коде).

Неправильно:

```
movlw   b'00000000'
movwf   trisb
...
bsf     portb, 1              ; Включить реле
...
movlw   .3                    ; Ошибка данных в EEPROM, мигнуть 3 раза
call    BlinkLeds
...
movlw   .50                   ; Ждем 50 миллисекунд
movwf   wait_timer
```

Правильно:

```
TRISB_CONST           equ     b'00000000'
BLINKS_EEPROM_ERROR   equ     .3
WAIT_TIME             equ     .50

#define     PIN_RELAY         portb, 1
...
...
movlw   TRISB_CONST
movwf   trisb
...
bsf     PIN_RELAY
...
movlw   BLINKS_EEPROM_ERROR
call    BlinkLeds
...
movlw   WAIT_TIME
movwf   wait_timer
```

Такой подход позволяет быстро производить настройку программы или менять ее параметры (например, перенос вывода, управляющего реле, на RB2 потребует замены всего двух констант: TRISB_CONST и PIN_RELAY).

Однако, есть константы, которые именовать не нужно ввиду их однозначности, например:

- В минуте всегда 60 секунд
- В байте 8 бит
- При переводе в BCD-код всегда оперируем со степенью числа 10
- И т.д.

6.2 Соблюдение системы счисления

Очень распространенной является концептуальная ошибка, при которой константа задается в несоответствующей ей по смыслу системе счисления. Даже если в комментариях поясняется ее истинное значение, это, во-первых, очень некрасиво, а во-вторых, создает благоприятные условия для совершения ошибок в будущем (например, в уме можно нечаянно неправильно перевести из одной системы в другую; или, исправив константу в коде, забывают поправить комментарий – получается разночтение, при котором программист начинает путаться: а какое из значений верное?). Например:

Неправильно:

```
        movlw   .65                     ; .65 = 0x41 = 01000001
        movwf   trisb                   ; RB0, RB6 – входы, остальные - выходы
        ...
        movlw   0x9E                    ; Запускаем таймер на 25000 отсчетов
        movwf   tmr1h
        movlw   0x58
        movwf   tmr1l
```

Правильно:

```
#define TRISB_CONST     b'01000001'     ; RB0, RB6 – входы, остальные - выходы
#define TMR1_CONST      .25000          ; Период для TMR1
        ...
        movlw   TRISB_CONST
        movwf   trisb
        ...
        movlw   high(-(TMR1_CONST))
        movwf   trm1h
        movlw   low(-(TMR1_CONST))
        movwf   tmr1l
```

Ассемблеру MPASM присуща одна проблема: по умолчанию он числовые константы воспринимает как шестнадцатеричные. Это не проблема для тех программистов, кто кроме PIC'ов и MPLAB'а ни с чем не работает. Но для тех, кому приходится иметь дело с ассемблерами для контроллеров нескольких производителей, это головная боль. Дело в том, что в подавляющем большинстве трансляторов принято, что отсутствие явного указания системы счисления говорит о том, что число записано в десятичной. Но не для MPASM'а.

Я бы очень рекомендовал при задании любой константы явно указывать ее вид: точка перед числом для десятичных чисел, префикс 0x или суффикс h для шестнадцатеричных и т.п. Напомню, что в начало каждого файла можно добавлять директиву:

```
        radix dec
```

Эта директива скажет ассемблеру о том, что по умолчанию нужно использовать десятичную систему счисления. Пока кто-то из вас работает только с PIC'ами, это кажется несущественным. Но если случится так, что надо будет расти и расширять свой кругозор, осваивая новые платформы и новые системы команд, то привычка писать правильно сыграет вам на руку.

6.3 Осмысленные значения констант

Константам следует задавать осмысленные значения. Т.е.:

- Время задавать в секундах (или миллисекундах), а не в периодах переполнения TMR1 = 65536 мкс
- Напряжение задавать в вольтах (или милливольтах, т.к. ассемблер не поддерживает вещественные числа), а не в единицах младшего разряда АЦП
- Температуру – в градусах
- Частоту – в Герцах
- И т.д.

Неправильно:

```
#define U_POROG    .102          ; Напряжение порога на входе АЦП
...
call    ReadADC                  ; получить текущее значение напряжения на входе АЦП
sublw   U_POROG                  ; сравнить с порогом
btfss   status, C
...
```

В данном случае совершенно непонятно, какое напряжение проверяется. Если на момент написания программы программист это еще будет помнить, то по прошествии года-двух ему придется все пересчитывать, чтобы вспомнить, что он с чем сравнивал.

Правильно:

```
#define U_POROG   .2000         ; Напряжение порога в милливольтах
#define U_REF     .5000         ; Величина опорного напряжения для АЦП в милливольтах
#define ADC_WIDTH .8

...

call    ReadADC
sublw   U_POROG*(1<<ADC_WIDTH)/U_REF
btfss   status, C
...
```

7 Комментирование

Комментирование – это отдельная тема для разговора. Многие программисты, заглядывая даже в собственные исходники, написанные более года назад, испытывают большие трудности с вниканием не только в логику работы программы, но и в логику некоторых алгоритмических решений и вывертов. Я уже не говорю, насколько сложно разбираться в плохо комментированной чужой программе.

Комментирование – это больной вопрос. Программисты, не умеющие комментировать, как правило, совершают три ошибки:

- Не пишут комментарии вообще
- Пишут так много комментариев, что они разбавляют текст программы и не дают его воспринять как единое целое
- Пишут бессмысленные комментарии

7.1 Почему не пишут комментарии?

- «Время поджимает, писать некогда»
- «Это временный код, его не нужно комментировать»
- «Я и так все запомню»
- «Моя программа понятна и без комментариев»
- «В код, кроме меня, никто не заглядывает»
- «Комментарии делают текст пестрым и затрудняют чтение самой программы»
- «Я потом откомментирую»

Все эти отговорки можно назвать одним словом – безобразие.

7.2 Содержание комментариев

Часто вижу, что программисты просто не умеют писать комментарии. Они знают, что комментарий просто обязан быть, поэтому пишут его только для того, чтобы он там был, особенно не задумываясь о его содержательной части. Доходит даже до комичных ситуаций, когда автор подробно комментирует каждую строку вплоть до того, что в комментариях расшифровывает значение мнемоник инструкций ассемблера. Такие комментарии порой хуже, чем их отсутствие.

Программисты должны понимать, что комментарий в первую очередь пишется для себя, для того, чтобы в программе было легче разбираться и ориентироваться, чтобы пояснить тонкости какого-то программного решения, а не для того, чтобы на протяжении 10 лет напоминать самому себе, что делает инструкция **movlw**.

Неправильно:

```
        movlw   .102            ; Заносим в аккумулятор число 102
        subwf   adresh, w       ; Вычитаем аккумулятор из регистра adresh
                                ; Результат помещаем в аккумулятор
        btfss   status, C       ; Проверяем бит переноса
        goto    EXIT            ; Если он сброшен, то переходим на метку EXIT
        ...
```

Смешно? Тем не менее, такое встречается. Но что мы поняли из этих комментариев? Ровным счетом – ничего, т.к. они просто-напросто дублируют текст программы, т.е. в них написано то, что человек проговаривает про себя, когда читает текст программы.

Правильно:

```
movlw   .102          ; Не делать калибровку, если напряжение аккумулятора ниже 2В
subwf   adresh, w     ;
btfss   status, C     ;
goto    Exit          ;
...
```

Этого комментария будет достаточно. Обратите внимание, что если одна строка комментария относится сразу к нескольким инструкциям, то все инструкции дополняются пустым комментарием, чтобы визуально выделить блок, к которому относилось пояснение.

7.3 Что должно быть в комментариях

- Спецификация подпрограммы: краткое описание (что делает), список параметров и возвращаемое значение
- Назначение объявляемой переменной или константы
- Краткое, емкое, безызбыточное описание действия или пояснение к нему
- Пометки об изменениях в файле
- Указание отладочных узлов и временных конструкций

7.4 Чего в комментариях быть не должно

- Эмоций
- Описания устаревших действий
- Пояснений стандартных (тривиальных) действий, например, не нужно комментировать сохранение/восстановление контекста в обработчике прерываний
- Дублирования описания мнемоники
- Бесполезной информации
- Непонятных сокращений и не относящегося к специфике устройства жаргона
- Ложной и вводящей в заблуждение информации

7.5 Разделительный комментарий

В программе удобно пользоваться разделительными комментариями для отделения логически различных фрагментов кода. Примеры таких разделителей:

```
;-----------------------------------------------------------------------------
...
;=============================================================================
...
;*****************************************************************************
```

Часто различными разделителями пользуются в разных случаях. Например, для отделения функций друг от друга пользуются более заметной разметкой в виде звездочек, а для отделения блоков инструкций внутри функции – в виде дефисов. Это кому как удобнее.

8 Опасный код

Хочу предостеречь программистов, особенно начинающих, от опасных конструкций в программах, написанных на ассемблере. Крайне не рекомендуется использовать описанные ниже приемы:

8.1 goto $+n

Данная конструкция иногда используется программистами, чтобы не определять лишний раз метку при совершении перехода в пределах 5-10 инструкций. Чем этот прием может быть опасен? Рассмотрим пример:

Неправильно:

```
        movf    command, w          ; Если пришла команда CMD_SHIFT_LEFT
        xorlw   CMD_SHIFT_LEFT      ; то сдвинуть массив данных на 1 бит влево
        btfss   status, Z
        goto    $ + 5

        rlf     array + 0, f
        rlf     array + 1, f
        rlf     array + 2, f
        rlf     array + 3, f

; должны прыгнуть сюда
```

Через пару дней во время отладки кода программист выясняет, что он забыл перед сдвигом обнулить флаг переноса, из-за чего в массив **array** все время вдвигался мусор. Он это обнуление и добавил, забыв скорректировать шаг перехода в команде goto:

```
        movf    command, w          ; Если пришла команда CMD_SHIFT_LEFT
        xorlw   CMD_SHIFT_LEFT      ; то сдвинуть массив данных на 1 бит влево
        btfss   status, Z
        goto    $ + 5

        bcf     status, C
        rlf     array + 0, f
        rlf     array + 1, f
        rlf     array + 2, f
        rlf     array + 3, f        ; Ç А ПРЫГАЕМ СЮДА!

; хотим прыгнуть сюда
```

Правильно:

```
        movf    command, w          ; Если пришла команда CMD_SHIFT_LEFT
        xorlw   CMD_SHIFT_LEFT      ; то сдвинуть массив данных на 1 бит влево
        btfss   status, Z
        goto    SkipShiftLeft

        rlf     array + 0, f
        rlf     array + 1, f
        rlf     array + 2, f
        rlf     array + 3, f

SkipShiftLeft:
```

Данный подход исключает возможность совершения подобной ошибки в будущем.

Еще менее заметная ошибка:

Неправильно:

```
        btfss   status, Z
        goto    $+3
        banksel counter
        incf    counter, f

; хотим прыгнуть сюда
```

Таким кодом можно успешно пользоваться много лет, что вызовет у программиста иллюзию его абсолютной надежности. Но такой код будет работать только когда используется контроллер с двумя банками оперативной памяти, т.е. когда директива banksel разворачивается в 1 инструкцию установки/сброса бита RP0.

```
        btfss   status, Z
        goto    $+3
        bsf     status, RP0
        incf    counter, f

; Прыгнем сюда
```

Как только будет взят контроллер с тремя или четырьмя банками ОЗУ, программа начнет давать сбои.

```
        btfss   status, Z
        goto    $+3
        bsf     status, RP0
        bcf     status, RP1
        incf    counter, f                ; Ç А ПРЫГАЕМ СЮДА!

; хотим прыгнуть сюда
```

Правильно:

```
        btfss   status, Z
        goto    SkipIncCounter

        banksel counter
        incf    counter, f

SkipIncCounter:
```

Еще одна серьезная ошибка применения $ с командой goto: применение одной инструкции “goto $+1” вместо двух NOP'ов.

Неправильно:

```
        goto $+1      ; Задержка в 6 тактов
        goto $+1
        goto $+1
```

Дело в том, что этот код, успешно работавший на PIC16, при переносе на PIC18 ведет себя иначе: в PIC18 адресация ROM – побайтовая, и в команде goto младший бит не участвует, следовательно, мы получим эквивалент goto $, т.е. просто повиснем.

Тем не менее, есть случай, когда $ может быть применен в качестве аргумента инструкции goto – это «goto $» - единственный способ произвести программный сброс в контроллерах PIC12/PIC16 при включенном сторожевом таймере. Во всех остальных случаях применение такой конструкции чревато.

8.2 Использование псевдокоманд

PIC16 обладает очень скромным набором инструкций, и некоторые тривиальные операции иногда приходится расписывать двумя-тремя инструкциями. Типичный пример – условный переход. Для того, чтобы выполнить переход по условию ZERO=1, нужно написать две команды:

```
btfsc   status, Z
goto    Label
```

Разработчики MPASM решили упростить жизнь программистам, добавив так называемые псевдокоманды для повышения наглядности кода, некоторые из которых являются комбинацией двух команд (а некоторые – даже трех). В частности, вместо двух инструкций предлагается использовать одну:

```
BZ      Label       ; BZ расшифровывается как “Branch on Zero”
```

Данная псевдокоманда является краткой и наглядной. При ассемблировании она разворачивается в те же две команды: проверка флага Z и безусловный переход на метку Label.

MPASM предоставляет почти 30 таких псевдокоманд. Среди них и упомянутые уже условные переходы, и сброс/установка флагов АЛУ (CARRY, ZERO и пр.), и сложение с учетом переноса и т.д. (полный перечень есть в документации на MPASM в разделе “Instruction Sets”). Некоторые программисты активно пользуются подобными командами, делая свой код нагляднее.

Однако при использовании этих псевдоинструкций программиста поджидают опасности, которые приводят к очень трудноуловимым ошибкам. Привычка использовать псевдоинструкции стирает в понимании программиста грань между инструкциями контроллера и макросами, введенными разработчиками ассемблера для удобства. Рассмотрим для примера одну из самых распространенных псевдоинструкций **lgoto**. Эта псевдоинструкция выполняет переход на указанную метку с предустановкой битов 3 и 4 регистра **PCLATH** (для совершения переходов за пределы 2кб страниц памяти), т.е., записав:

```
lgoto   Label
```

в программе вы получите (допустим, метка Label находится во второй странице ROM):

```
bcf     pclath, 3
bsf     pclath, 4
goto    Label
```

Это очень удобно, т.к. во-первых, при чтении исходного текста программы мы анализируем всего одну инструкцию вместо трех, а во-вторых, нам не нужно постоянно помнить, где находится метка Label. Но привычка частого использования такой псевдоинструкции может привести к ошибке.

Неправильно:

```
        btfss   status, C
        lgoto   Label
```

Данная запись выглядит просто и понятно, однако, она ошибочна, т.к. данная запись развернется в:

```
        btfss   status, C
        bcf     pclath, 3           ; Пропущена будет только эта инструкция!
        bsf     pclath, 4
        goto    Label
```

Как видите, при любом значении флага CARRY будет произведен переход (только в случае CARRY=1 можно будет улететь вообще неизвестно куда, т.к. бит 3 регистра **PCLATH** не установится в нужное значение, и если там была «1», то программа попадет совсем в другую страницу). Причем такие ошибки почти незаметны при визуальном анализе кода. Хорошо, если программа небольшая и есть возможность пройти ее в симуляторе по шагам. Но в сильно разросшейся программе выловить такую ошибку бывает очень непросто.

Правильно:

```
        pagesel Label               ; Сперва формируем значения битов 3 и 4 регистра PCLATH
        btfss   status, C           ; И только потом делаем проверку и переход
        goto    Label
        pagesel $                   ; Не забываем восстанавливать биты 3 и 4 регистра PCLATH,
                                    ; если условие не было выполнено
```

Другими словами, псевдокомандами стоит пользоваться с большой осторожностью. Желательно выбрать из них две-три наиболее актуальных (таких как lgoto, lcall) и прочно забить себе в голову, что эти команды нельзя использовать совместно с условными инструкциями: btfss, btfsc, decfsz и пр.

8.3 Неполный перечень операндов

MPASM позволяет указывать не все операнды некоторых инструкций, дополняя неуказанные поля значениями по умолчанию. Я бы рекомендовал указывать все операнды в полной мере для всех инструкций. Это позволит избежать неправильного толкования поведения программы при визуальном анализе.

Неправильно:

```
        movf    array
        ...
        decfsz  Counter
```

Правильно:

```
movf    array, f
...
decfsz  Counter, f
```

При работе с PIC18 настоятельно рекомендуется указывать значение бита ACCESS:

Неправильно:

```
movf    array
movwf   postinc0
```

Правильно:

```
movf    array, f, s
movwf   postinc0, a
```

9 О прерываниях

9.1 Зачем нужен модуль прерываний

Вы знаете, что модуль прерываний – это блок в составе микроконтроллера, который при возникновении определенного события (переполнения таймера, изменения логического уровня на внешнем входе, сигнала от какого-либо периферийного модуля и т.п.) прерывает выполнение текущей программы, передает управление специальной подпрограмме (обработчику прерываний), которая обрабатывает произошедшее событие, и затем возвращает управление обратно прерванной программе.

Как ни странно, часто встречаю некорректное использование модуля прерываний некоторыми программистами. То есть, что значит некорректное? С точки зрения архитектуры контроллера, взаимодействия его регистров, стека и периферии оно допустимо, но сам факт некорректного применения настолько снижает эффективность и полезность этого модуля, что при разрастании программы до среднего уровня сложности у контроллера начинается нехватка ресурсов. Ниже я перечислю, что должно быть в обработчике прерываний и чего там быть не должно. Хоть большинство описанных ниже правил носит чисто рекомендательный характер и неследование им не приведет к неработоспособности программы, все они продиктованы опытом и четким пониманием назначения модуля прерываний.

9.2 Характеристики обработчика прерывания

При входе должен сохранять контекст, а при выходе восстанавливать его

Для контроллеров PIC12/PIC16/PIC18 обязательными к сохранению являются два регистра: **WREG** и **STATUS**. Остальные – в зависимости от используемого контроллера и функционала обработчика прерываний. В частности для контроллеров PIC16 с более чем одной страницей памяти программ требуется сохранять регистр **PCLATH**. Для контроллеров PIC18 часто требуется сохранять регистр **BSR**, если предполагается доступ к переменным в BANKED-области ОЗУ.

Стоит добавить, что в приоритетном режиме работы модуля прерываний в PIC18 для высокоприоритетного прерывания регистры **WREG**, **STATUS** и **BSR** сохранять не нужно (они сохраняются автоматически, а восстанавливаются инструкцией «retfie fast»).

При оформлении кода по сохранению контекста его обычно не комментируют, особенно, если он имеет стандартный набор регистров для сохранения.

*Примечание: в редчайших случаях какой-то из регистров можно не сохранять (например, иногда обработчик прерывания пишется без участия **WREG**), вот в таких случаях следует очень тщательно откомментировать причины несохранения и требования к обработчику. Но эти случаи настолько редкие, что бывают в жизни далеко не каждого профессионала, т.к. они сопровождают очень специфичные задачи. Заострять внимание на них не будем.*

Должен выполняться за минимальное время

Модуль прерываний является незаменимым помощником в реализации реакций на событие в реальном времени. При возникновении очередного прерывания во время работы подпрограммы-обработчика, оно будет отложено до завершения обработки текущего прерывания. Чем быстрее будет выполнен код обработки текущего, тем скорее будет обработано новое, тем, соответственно быстрее будет реакция на событие. В целом это определяет быстродействие создаваемого вами устройства (существует понятие детерминированности реакции на событие, определяемое, как гарантия реакции на событие в течение времени, не превышающего какую-то константу, вне зависимости от текущего состояния программы; чем это время меньше, тем лучше).

Должен обрабатывать только активные (xxIE = «1») прерывания

Микроконтроллеры PIC16 имеют всего один вектор прерывания (PIC18 – два), что вынуждает программиста одним обработчиком обрабатывать прерывания, возникшие от нескольких источников. Т.е., попадая в обработчик, требуется по очереди перебрать все предусмотренные программой флаги, чтобы выяснить, какое именно прерывание произошло, и обработать именно его. Здесь кроется небольшой подвох, связанный с тем, что некоторые источники прерываний могу временно запрещаться.

Типичный пример – прерывание по завершению передачи байта через USART. Флаг TXIF устанавливается сразу же при установке бита разрешения передачи (TXEN), т.е. тогда, когда еще ни один байт не передан. Поэтому, если программой предусмотрена отправка данных через USART из обработчика прерывания (кстати, очень правильный подход), то когда данных на передачу нет, чтобы прерывание не генерилось вхолостую, его запрещают TXIE = 0 (напомню, что бит события TXIF не может быть сброшен вручную; он сбрасывается автоматически только при записи данных в регистр передачи TXREG).

Неправильно:

```
      CODE    0x0004

Interrupt:
      ;... сохранение контекста

;-----------------------------------------------------------
; Прерывание по таймеру 0
;-----------------------------------------------------------
T0IF_Check:

      btfss   intcon, T0IF          ; было ли прерывание по таймеру?
      goto    T0IF_Skip

      bcf     intcon, T0IF

      ;...
      ;... код обработки
      ;...

T0IF_Skip:
;-----------------------------------------------------------
; Прерывание по завершению передачи USART
;-----------------------------------------------------------
TXIF_Check:

      btfss   pir1, TXIF            ; Даже при TXIE=0 программа попадет в обработчик
      goto    TXIF_Skip
      ;...
      ;... код обработки
      ;...

TXIF_Skip:
```

Допустим, в приведенном выше примере в обработчик попали по прерыванию при переполнении TMR0. Т.е. пока этого прерывания не было, мы в обработчик не попадали даже при установленном TXIF, т.к. бит разрешения был сброшен (TXIE = 0). Но все попадания внутрь обработчика будут вызывать одновременно и обработку TXIF.

Правильно:

```
        CODE    0x0004

Interrupt:
        ;... сохранение контекста

;-----------------------------------------------------------
; Прерывание по таймеру 0
;-----------------------------------------------------------
T0IF_Check:

        btfsc   intcon, T0IF          ; было ли прерывание по таймеру?
        btfss   intcon, T0IE          ; разрешено ли прерывание?
        goto    T0IF_Skip

        bcf     intcon, T0IF

        ;...
        ;... код обработки
        ;...

T0IF_Skip:
;-----------------------------------------------------------
; Прерывание по завершению передачи USART
;-----------------------------------------------------------
TXIF_Check:

        banksel pie1
        btfss   pie1, TXIE
        goto    TXIF_Skip

        banksel pir1
        btfss   pir1, TXIF            ; Теперь на эту проверку попадем только при TXIE = 1
        goto    TXIF_Skip
        ;...
        ;... код обработки
        ;...

TXIF_Skip:
        banksel ...
```

Должен максимально быстро сбрасывать флаг, вызвавший прерывание

На практике столкнулся с тем, что многие неправильно понимают термин «отложенное прерывание». Почему-то некоторые считают, что где-то в недрах контроллера есть большой резервуар для запоминания того, какие прерывания и по сколько раз произошли, пока их программа формировала задержку в теле прерывания (естественно, при сброшенном GIE). И им кажется, что как только прерывания будут разрешены, то они возникнут столько раз, сколько возникло событий. Например, полагают, что если во время GIE=0 на RB0 пришло 5 импульсов, то после установки GIE=1 и прерывание будет сгенерировано 5 раз. **Это не так!** «Откладываться» может только факт возникновения события (в виде установленного флага xxIF), но не количество этих событий. Таким образом, отложенным прерыванием считается прерывания от одного источника, возникшее во время обработки прерывания от другого (или во время, когда прерывания были запрещены по каким-то другим причинам) и не имеющее возможности быть обработанным немедленно. Количество отложенных прерываний соответствует количеству возможных источников прерываний, но откладывается не более, чем по одному событию на источник.

Поэтому, если с момента возникновения события, вызывающего прерывания, до момента сброса соответствующего ему флага это же событие происходит еще раз, то оно будет обработано только единожды. Следовательно, флаг прерывания нужно сбрасывать максимально быстро:

Неправильно:

```
	CODE	0x0004

Interrupt:
	;... сохранение контекста

;-----------------------------------------------------------
; Внешнее прерывание
;-----------------------------------------------------------
INTF_Check:

	btfsc	intcon, INTF		; было ли прерывание
	btfss	intcon, INTE		; разрешено ли прерывание
	goto	INTF_Skip

	;...
	;... код обработки
	;...

	bcf	intcon, INTF		; Сброс события происходит затянуто

INTF_Skip:
```

Недостаток такого подхода в том, что неоправданно затянуто время с момента возникновения прерывания, до момента сброса флага INTF. Чем это время больше, тем выше вероятность пропустить два быстро идущих друг за другом одинаковых события.

Правильно:

```
	CODE	0x0004

Interrupt:
	;... сохранение контекста

;-----------------------------------------------------------
; Внешнее прерывание
;-----------------------------------------------------------
INTF_Check:

	btfsc	intcon, INTF		; было ли прерывание
	btfss	intcon, INTE		; разрешено ли прерывание
	goto	INTF_Skip

	bcf	intcon, INTF		; первым делом сбрасываем событие, а потом обрабатываем

	;...
	;... код обработки прерывания
	;...

INTF_Skip:
```

9.3 Чего не должно быть в обработчике прерывания

Длительно выполняющегося кода

Собственно, причины, почему так делать не стоит, мы уже обсудили. Выполняя в обработчике прерываний длительные операции, вы лишаете контроллер способности решать важнейшую задачу – быстро реагировать на события. Кроме того, помимо замедленной реакции, вы столкнетесь с пропуском событий, если одно и то же событие успеет произойти два и более раз.

Тем не менее, иногда у начинающих встречается в обработчиках прерываний то код задержки на 20 мс, то ожидание завершения записи в EEPROM, то полный цикл АЦ-преобразования. **Настоятельно рекомендую не использовать длительно выполняющийся код в теле подпрограммы обработчика прерываний**.

Вызовов подпрограмм

В принципе, для программ, написанных на ассемблере, рекомендация спорная. Но ассемблер для многих – первый шаг, дальше пойдут языки высокого уровня, где немного другие правила взаимодействия программиста с контроллером. Почему нельзя этого делать? Есть несколько причин:

1. В PIC16 всего 8-уровневый стек. Вызывая подпрограмму из обработчика, вы лишаете свою программу одного уровня вложенности. При разрастании программы это может сказаться. Если вы пишете модульную программу, то вам уровень вложенности вообще может быть неизвестен (например, используете библиотеку, поставляемую без исходных текстов), и рисковать одним уровнем вложенности не стоит. Кроме того, придется постоянно быть уверенным в том, что вызываемая подпрограмма сама ничего не вызывает, иначе теряете уже не один уровень вложенности.
2. Добавлением вызова подпрограммы в обработчик вы переводите алгоритм своей программы на другой уровень абстракции. Т.е. надо будет либо заботиться уже и о том, чтобы вызываемая программа была недоступной из основной программы, либо обеспечивать реентерабельность вызываемой функции, что средствами ассемблера не очень удобно (хотя и возможно). Недопонимание механизма реентерабельности может привести к порче данных, которыми оперирует вызываемая функция.

Глобального разрешения прерываний (установки флага GIE)

Некоторые позволяют себе пользоваться достаточно опасным приемом: при входе в обработчик прерывания устанавливают бит GIE, тем самым разрешая модулю прерываний еще раз прервать подпрограмму и произвести повторный вход в обработчик, обеспечивая как бы вложенные обработчики прерываний. Хоть я и видел пару раз удачные реализации такого подхода, все-таки предостерегу вас, особенно тех, кто еще начинающий, от такого подхода. Он требует не только недюжих знаний архитектуры контроллера, но и превосходного алгоритмического мышления и умения строить абстрактные модели.

Поэтому, если вы начинающий, но вам кажется, что вложенные обработчики помогут вам решить конкретную задачу, значит, вы неправильно к ней подошли. Ищите другое решение.

9.4 Как правильно запрещать/разрешать прерывания в программе

Часто в чужих программах вижу, как прерывания запрещаются и восстанавливаются прямой записью «0» и «1», соответственно, в бит GIE. Чем это плохо? В общем случае неизвестно, были ли разрешены или запрещены прерывания на момент выполнения текущего запрета. Если они были запрещены, и мы их опять запретили повторной записью «0» в бит GIE, то ничего страшного не произойдет. Но при восстановлении безусловной записью «1» в GIE – произойдет. Т.к. предполагалось, что прерывания изначально были отключены, то и восстанавливать следует отключенное состояние.

Неправильно:

```
bcf     intcon, GIE
...
...
bsf     intcon, GIE
```

Для того, чтобы не возникало коллизий, перед запретом текущее состояние бита GIE требуется сохранить, а при восстановлении воспользоваться сохраненным значением.

Правильно:

```
        movf    intcon, w        ; Копируем содержимое регистра во временную переменную
        movwf   intcon_Temp
        bcf     intcon, GIE      ; Запрещаем прерывания

        ...
        ...
        btfsc   intcon_Temp, GIE ; Восстанавливаем прерывания
        bsf     intcon, GIE
```

Если на время запрета прерываний есть уверенность в том, что флаг CARRY не затрагивается, то можно применить следующий прием (однако с ним нужно быть осторожным и применять его только на очень коротких участках кода):

Правильно:

```
        rlf     intcon, w        ; Выдвигаем бит GIE в CARRY, регистр intcon не меняется
        bcf     intcon, GIE      ; Запрещаем прерывания

        ...
        ...
        btfsc   status, C        ; Восстанавливаем прерывания
        bsf     intcon, GIE
```

Таким образом можно сэкономить одну ячейку памяти.

9.5 Доступ к переменным внутри прерывания

Обеспечение атомарного доступа – это задача, присущая не только и не столько ассемблеру, сколько параллельно выполняющимся процессам. Опишу проблему в двух словах. Допустим, у вас в программе есть двухбайтовая переменная в которую вы в прерывании сохраняете измеренное значение АЦ-преобразования:

```
;----------------------------------------------------------
; Определение переменных (фрагмент)
;----------------------------------------------------------

        cblock 0x20
        ...
        adc_data:2
        ...
        endc

        ...

;----------------------------------------------------------
; Прерывание по завершению АЦ-преобразования
;----------------------------------------------------------
ADIF_Check:

        banksel pie1                ; Пропускаем, если прерывание запрещено
        btfss   pie1, ADIE
        goto    ADIF_Skip

        banksel pir1                ; Пропускаем, если АЦ-преобразование не завершено
        btfss   pir1, ADIF
        goto    ADIF_Skip

        bcf     pir1, ADIF
        ;----------------------------------------------------------
```

```
        movf    adresh, w            ; adc_data = adresh:adresl
        movwf   adc_data+1

        banksel adresl               ; младший байт
        movf    adresl, w
        banksel adc_data
        movwf   adc_data

        banksel adcon0
        bsf     adcon0, GO           ; Запускаем следующее преобразование

ADIF_Skip:
        ...
```

Где-то в программе вам понадобилось сравнить это значение с константой.

Неправильно:

```
        #define U_POROG .300         ; Для наглядности данного примера я привожу константу
                                     ; в явном виде, т.е. в единицах АЦП

;---------------------------------------------------------
; Выполняем сравнение:  adc_data с U_POROG
;---------------------------------------------------------
CompareADCData:

        movlw   high(U_POROG)                 ; Сравниваем старший байт
        subwf   adc_data+1, w
        btfss   status, C
        goto    CAD_Less                      ; Меньше

        btfss   status, Z
        goto    CAD_Greater                   ; Больше

        ; Сюда попадаем, когда старшие байты равны

        movlw   low(U_POROG)                  ; Сравниваем младший байт
        subwf   adc_data+0, w
        btfss   status, C
        goto    CAD_Less                      ; Меньше

        btfss   status, Z
        goto    CAD_Greater                   ; Больше

;---------------------------------------------------

CAD_Equ:
        retlw   0

CAD_Less:
        retlw  -1

CAD_Greater:
        retlw   1
```

Теперь рассмотрим ситуацию, когда значение напряжения на входе АЦП близко к Vdd/2. Из-за шума от источника питания у результата преобразования АЦП будет небольшой шум в младших разрядах. Т.к. напряжение близко к Vdd/2, то результат преобразования будет в районе 512 +/- 2 единицы младшего разряда, т.е. меняться от 510 до 514. Здесь и кроется проблема. В шестнадцатеричном представлении эти границы будут выглядеть так: 0x1FE..0x202. Как видите от измерения к измерению старший байт результат будет менять свое значение: то 1 то 2.

Теперь представьте, что сравнение началось тогда, когда adc_data была равна 0x1FF. Нам нужно сравнить его с U_POROG = 300 = 0x12C. Понятно, что 0x1FF > 0x12C, поэтому функция должна нам вернуть 1 (метка CAD_Greater).

Возможный вариант действия программы:

1. Начинаем сравнивать старший байт 0x01 со старшим байтом **U_POROG** = 0x1
2. Т.к. они равны, то в соответствии со спецификацией инструкции SUBWF, флаги будут C=1 и Z=1.
3. После этого программа будет сравнивать младшие байты.
4. Где-то в промежутке между двумя обращениями к переменной **adc_data** (выделено жирным) произошло прерывание, в котором переменная adc_data обновилась и стала равной 0x200. Т.е. результат АЦ-преобразования изменился всего на одну единицу младшего разряда.
5. Программа начинает сравнивать младшие байты. Но младший байт **adc_data** теперь равен 0x00, поэтому произойдет сравнение 0x00 с 0x2C
6. Очевидно, что после этого сравнения программа решит, что **adc_data** < **U_POROG** и возвратит «–1» (метка CAD_Less).

Как видите, даже при том, что оба значения, которые принимала переменная **adc_data** во время сравнения (0x1FF и 0x200) больше порогового (0x12C), функция решила, что **adc_data** < **U_POROG**. Это произошло из-за того, что мы не обеспечили атомарный доступ к переменной, изменяющейся в параллельном процессе (атомарный = неделимый). Как быть? Есть два варианта.

Первый - запрещать прерывания на все время сравнения.

Правильно 1:

```
;--------------------------------------------------------
; Выполняем сравнение:  adc_data с U_POROG
;--------------------------------------------------------
CompareADCData:
      banksel pie1
      bcf     pie1, ADIE
      banksel adc_data

      ...
      ...
      ...

CAD_Equ:
      movlw   0
      goto    CAD_Exit

CAD_Less:
      movlw  -1
      goto    CAD_Exit

CAD_Greater:
      movlw   1

CAD_Exit:
      banksel pie1
      bsf     pie1, ADIE
      return
```

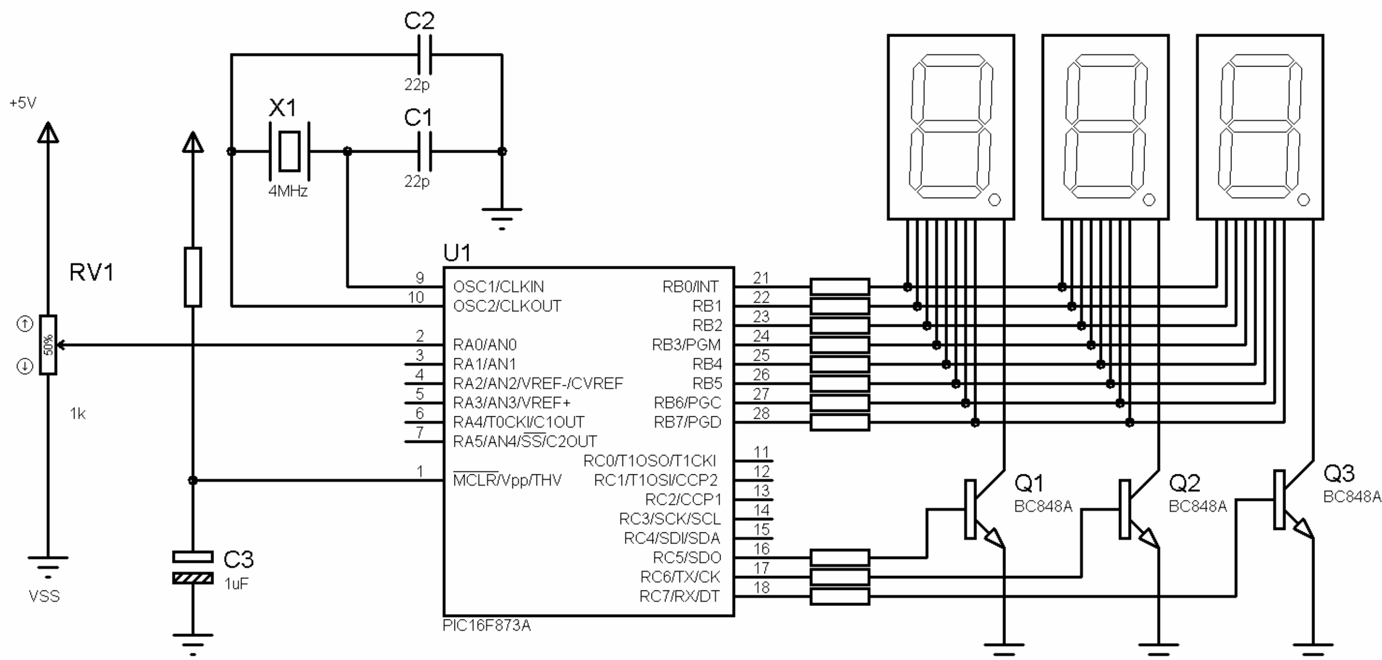
Второй - если есть 2 свободные ячейки памяти, запрещать прерывания только на короткий промежуток времени для копирования adc_data в промежуточную переменную, а затем производить сравнение U_POROG со значением промежуточной переменной

Правильно 2:

```
;-------------------------------------------------------
; Выполняем сравнение:  adc_data с U_POROG
;-------------------------------------------------------
CopareADCData:

      rlf     intcon, w                    ; запрещаем прерывание с запоминанием
      bcf     intcon, GIE                  ; текущего состояния GIE в CARRY

      movf    adc_data, w                  ; Выполняем копирование
      movwf   temp_data
      movf    adc_data+1, w
      movwf   Temparray+1

      btfsc   status, C                    ; Восстанавливаем GIE
      bsf     intcon, GIE

      movlw   high(U_POROG)                ; Сравниваем старший байт
      subwf   temp_data +1, w
      btfss   status, C
      goto    CAD_Less                     ; Меньше

      ...
      ...
      ...
```

10 Пример хорошо оформленной программы

Для примера я написал небольшую программу «Вольтметр». Программа измеряет напряжение на аналоговом входе раз в 1мс, вычисляет среднее арифметическое последних 64 выборок и выводит получившееся значение на индикатор с точностью до сотых долей вольта.



Примечание: для демонстрации наглядности специально не применялась цветовая подсветка синтаксиса и особые параметры шрифтов.

```
;*****************************************************************************
; voltmetr.asm
;
; Одноканальный вольтметр 0..Vdd с выводом на 3х позиционный 7-сегментный индикатор.
;
; RA0 - аналоговый вход
; RC5, RC6, RC7 - общие катоды сегментов
; portb - управление сегментами
;
; Программа является примером, поясняющим правила оформления, описанных в статье:
; "MPASM: как правильно оформлять программы на ассемблере для PIC-контроллеров"
;
; Автор: В.Тимофеев, testerplus@mail.ru, osa@pic24.ru
;
; История:    13.11.2010 -      Файл создан
;
;*****************************************************************************

        list p=16f873a
        include <p16f873a.inc>

        __CONFIG _HS_OSC & _CP_ALL & _LVP_OFF & _DEBUG_OFF & _WDT_ON & _BODEN_OFF

        radix dec

        ERRORLEVEL -302


;*****************************************************************************
;
;       КОНСТАНТЫ ПОЛЬЗОВАТЕЛЯ (могут меняться в зависимости от режима)
;
;*****************************************************************************

F_OSC           equ     .4000000        ; Тактовая частота (Гц)
F_ADC           equ     .1000           ; Частота измерений ADC
                                        ; Для отображения на индикаторе производится
                                        ; осреднение по 128 выборкам

;-----------------------------------------------------------------------------
;       Параметры входного сигнала
;-----------------------------------------------------------------------------
V_REF           equ     .5000               ; Задаем напряжение опоры (мВ)
ADC_RESOLUTION  equ     .10                 ; Разрешение АЦП
AVERAGE_CONST   equ     .16-ADC_RESOLUTION  ; Константа осреднения будет зависеть от
                                            ; разрядности АЦП с тем, чтобы сумма
                                            ; измерений четко уложилась в два байта

;*****************************************************************************
;
;       КОНСТАНТЫ РАБОЧИЕ
;
;*****************************************************************************

;-----------------------------------------------------------------------------
;       Порты ввода вывода
;-----------------------------------------------------------------------------
TRISA_CONST     equ     b'000001'       ; AN0 - вход
PORTA_CONST     equ     b'000000'

TRISB_CONST     equ     b'00000000'     ; Все выходы
PORTB_CONST     equ     b'11111111'     ; Управление сегментами

SEG_A           equ     (1<<0)          ; Назначение сегментов в индикаторах
SEG_B           equ     (1<<1)          ;
SEG_C           equ     (1<<2)          ;    A
SEG_D           equ     (1<<3)          ;  F   B
SEG_E           equ     (1<<4)          ;    G
SEG_F           equ     (1<<5)          ;  E   C
SEG_G           equ     (1<<6)          ;    D   H
SEG_H           equ     (1<<7)          ;

TRISC_CONST     equ     b'00000000'     ; Все выходы
PORTC_CONST     equ     b'00000000'

#define PIN_DIGIT_0     PORTC, 5
#define PIN_DIGIT_1     PORTC, 6
#define PIN_DIGIT_2     PORTC, 7
```

```
;------------------------------------------------------------------------------
;       Параметры АЦП
;------------------------------------------------------------------------------
ADCON1_PCFG_CONST equ   b'1110'      ; RA0 - аналоговый, остальные - цифровые

;------------------------------------------------------------------------------
;       Параметры TMR0
;------------------------------------------------------------------------------
TMR0_PERIOD     equ     F_OSC/4/F_ADC
TMR0_PRS_CONST  equ     0x01
TMR0_CONST      equ     TMR0_PERIOD / (2 << TMR0_PRS_CONST)


; Проверка, что получили константу, соответствующую разрядности TMR8 (не более 8 бит)
#if TMR0_CONST >= 256
error Неправильно выбрана константа TMR0_PRS_CONTS! Следует увеличить ее значение!
#endif



;******************************************************************************
;
;       ПЕРЕМЕННЫЕ
;
;******************************************************************************

        CBLOCK  0x20
        output_text  :3             ; Текстовое представление напряжения
        cur_digit                   ; Текущая цифра для отображения

        work_data    :3             ; Для преобразования из единиц АЦП в милливольты
        adc_average_data :2         ; Результат АЦП
        adc_average_counter         ; Счетчик измерений при усреднении (см. ADC_AVERAGE)

        product     :3              ; Переменные для подпрограммы умножения
        multiplier  :2

        divider                     ; Переменная для подпрограммы деления со сдвигом

        div_result                  ; Переменные для bin2dec преобразования

        ENDC
;------------------------------------------------------------------------------

        CBLOCK  0x70
        wreg_temp                   ; Регистры для сохранения контекста
        status_temp
        fsr_temp
        pclath_temp
        ENDC


;******************************************************************************
;
;       ВЕКТОР СБРОСА
;
;******************************************************************************

        ORG     0x0000
        lgoto   Start


;******************************************************************************
;
;       ПРЕРЫВАНИЕ
;
;******************************************************************************
        ORG     0x0004
Interrupt:
        movwf   wreg_temp
        movf    status, w
        movfw   status_temp
        movf    pclath, w
        movwf   pclath_temp
        movf    fsr, w
        movwf   fsr_temp
        clrf    status
        clrf    pclath
```

```
;--------------------------------------------------------------------------------
; Прерывание TMR0: смена текущей позиции индикатора
;--------------------------------------------------------------------------------
T0IF_Check:

        btfsc   intcon, T0IF
        btfss   intcon, T0IE
        goto    T0IF_Skip
        ;------------------------------------------------------------------------
        bcf     intcon, T0IF            ; Обновляем таймер
        movlw  -TMR0_CONST              ;
        addwf   TMR0, f                 ;
        ;------------------------------------------------------------------------

        btfss   adc_average_counter, AVERAGE_CONST
                                        ; Пропускаем измерение, если основная программа
                                        ; еще не обработала данные
        bsf     adcon0, GO              ; Запускаем следующее преобразование

        bcf     PIN_DIGIT_0             ; Сначала гасим все три индикатора
        bcf     PIN_DIGIT_1             ;
        bcf     PIN_DIGIT_2             ;
        clrf    portb                   ;
        ;------------------------------------------------------------------------
        movf    cur_digit, w            ; Включаем один из сегментов
        btfsc   status, Z               ;
        bsf     PIN_DIGIT_0             ;
        btfsc   cur_digit, 0            ;
        bsf     PIN_DIGIT_1             ;
        btfsc   cur_digit, 1            ;
        bsf     PIN_DIGIT_2             ;
        ;------------------------------------------------------------------------
        addlw   output_text             ; Зажигаем нужные сегменты
        movwf   fsr                     ; fsr = адрес цифры для вывода
        movf    indf, w                 ;
        movwf   portb                   ;
        ;------------------------------------------------------------------------
        incf    cur_digit, f            ; Берем следующую позицию (0,1,2,0,1,2,0,...)
        movf    cur_digit, w            ;
        xorlw   0x3
        btfsc   status, Z               ;
        clrf    cur_digit               ;

T0IF_Skip:


;--------------------------------------------------------------------------------
; Прерывание АЦП: завершение АЦ-преобразования
;--------------------------------------------------------------------------------


ADIF_Check:

        banksel pie1                    ; Пропускаем, если прерывание запрещено
        btfss   pie1, ADIE              ; или не завершено
        goto    ADIF_Skip               ;
        banksel pir1                    ;
        btfss   pir1, ADIF              ;
        goto    ADIF_Skip               ;
        ;------------------------------------------------------------------------
        bcf     pir1, ADIF
        ;------------------------------------------------------------------------
                                        ; Условия сброса сторожевого таймера:
                                        ; 1. Прерываня T0IF и ADIF возникают и
                                        ;    обрабатываются не реже, чем раз в 18 мс
        clrwdt                          ; 2. Основная программа периодически получает
                                        ;    управление (т.к. adc_average_counter обнуляется)
                                        ; Следовательно, если попали в эту точку, значит,
                                        ; программа работает в штатном режиме
        ;-------------------------------;

        banksel adresl                  ; Выполняем сложение:
        movf    adresl, w               ; adc_average_data = adc_average_data + ADRES
        banksel adresh                  ;
        addwf   adc_average_data, f     ;
        movf    adresh, w               ;
        btfsc   status, c               ;
        incf    adresh, w               ;
        addwf   adc_average_data+1, f   ;
```

```
        incf    adc_average_counter, f              ; Увеличиваем счетчик выборок
        btfss   adc_average_counter, AVERAGE_CONST ; Собрали ли нужное количество выборок?
        goto    ADIF_Skip                           ; Еще нет.

        movf    adc_average_data, w      ; Если собрали, то переносим данные в рабочую
        movwf   work_data                ; переменную: work_data = adc_average_data
        movf    adc_average_data+1, w    ;
        movwf   work_data+1              ;
        clrf    work_data+2              ;

        clrf    adc_average_data         ; Готовим переменную к сбору следующих выборок
        clrf    adc_average_data + 1
        ;----------------------------------------------------------------------
ADIF_Skip:
        clrf    status

;------------------------------------------------------------------------------
Int_Exit:
        movf    fsr_temp, w
        movwf   fsr
        movf    pclath_temp, w
        movwf   pclath
        movf    status_temp, w
        movfw   status
        swapf   wreg_temp, f
        swapf   wreg_temp, w
        retfie


;******************************************************************************
;
;       ИНИЦИАЛИЗАЦИЯ
;
;******************************************************************************
Start:
        clrf    status
        clrf    intcon
        ;---------------------------------------------------------------------
        ; Настройка цифровых портов
        ;---------------------------------------------------------------------
        movlw   PORTA_CONST
        movwf   porta
        movlw   PORTB_CONST
        movwf   portb
        movlw   PORTC_CONST
        movwf   portc

        banksel trisa
        movlw   TRISA_CONST
        movwf   trisa
        movlw   TRISB_CONST
        movwf   trisb
        movlw   TRISC_CONST
        movwf   trisc
        ;---------------------------------------------------------------------
        ; Настройка аналогового входа
        ;---------------------------------------------------------------------
        banksel adcon1
        movlw   ADCON1_PCFG_CONST
        movwf   adcon1
        bsf     adcon1, ADFM             ; Правое выравнивание

        banksel adcon0
        clrf    adcon0                   ; Выбран канал 0 (AN0)
        bsf     adcon0, ADON             ; Включить модуль АЦП
        bsf     adcon0, ADCS1            ; Clock = Fosc/32

        ;---------------------------------------------------------------------
        ; Настройка таймера
        ;---------------------------------------------------------------------

        banksel option_reg
        movlw   TMR0_PRS_CONST
        movwf   option_reg

        banksel tmr0
        movlw   -TMR0_CONST
        movwf   tmr0
```

```
        ;---------------------------------------------------------------------
        ; Настройка прерываний
        ;---------------------------------------------------------------------

        bsf     intcon, T0IE
        banksel pie1
        clrf    pie1
        bsf     pie1, ADIE

        BSF     intcon, PEIE
        BSF     intcon, GIE
        clrf    STATUS




;*****************************************************************************
;
;       ОСНОВНОЙ ЦИКЛ
;
;*****************************************************************************


        clrf    adc_average_counter

        clrf    adc_average_data        ; Сумма для осреднения изначально = 0
        clrf    adc_average_data +1     ;
        clrf    adc_average_data +2     ;

        movlw   8                       ; При старте на дисплей выводится 888
        call    GetSegments             ;
        movwf   output_text             ;
        movwf   output_text+1           ;
        movwf   output_text+2           ;

;-----------------------------------------------------------------------------

Main:
        btfss   adc_average_counter, AVERAGE_CONST
        goto    Main

;-----------------------------------------------------------------------------

        bcf     adc_average_counter, AVERAGE_CONST

        ;---------------------------------------------------------------------
        ; Сейчас в рабочей переменной word_data сумма последних
        ; (1<<AVERAGE_CONST) измерений
        ;---------------------------------------------------------------------

        movlw   AVERAGE_CONST
        lcall   DivideShift

        ;---------------------------------------------------------------------
        ; Переводим из единиц АЦП в милливольты:
        ; work_data = work_data * V_REF/(1<<ADC_RES)
        ;---------------------------------------------------------------------
        call    MultiplyVRef            ; Умножить work_data на V_REF
        movlw   ADC_RESOLUTION
        lcall   DivideShift             ; Делить work_data на (1<<ADC_RESOLUTION)

        call    Bin2Dec                 ; Перевести work_data в массив десятичных чисел

        lgoto    Main
```

```
;**************************************************************************
;
;       ФУНКЦИИ
;
;**************************************************************************



;**************************************************************************
;       MultiplyVRef
;--------------------------------------------------------------------------
; Функция выполняет перемножение: work_data = work_data * (V_REF/10)
; Т.к. используются только 3 значащие цифры, то единицы мВ не нужны,
; поэтому пересчет ведется в измерении 10мВ
;
; Использует переменные:
;       multiplier[2] - второй множитель
;       product[3]    - произведение
; На выходе:
;       work_data = product
;--------------------------------------------------------------------------
MultiplyVRef:
        clrf    product                 ; Обнуляем произведение
        clrf    product+1               ;
        clrf    product+2               ;
        clrf    product+3               ;

        movlw   high(V_REF/.10)         ; Готовим второй множитель
        movwf   multiplier+1            ;
        movlw   low(V_REF/.10)          ;
        movwf   multiplier+0            ;

;------------------------------------------------------------------
MV_Loop:
        btfss   multiplier, 0           ; Если очередной бит
        goto    MV_SkipSumm             ; второго множителя установлен,

        movf    work_data, w            ; то выполняем сложение двух 3-байтовых
        addwf   product, f              ; чисел
        movf    work_data+1, w          ;
        btfsc   status, C               ;
        incfsz  work_data+1, w          ;
        addwf   product +1, f           ;
        movf    work_data+2, w          ;
        btfsc   status, C               ;
        incfsz  work_data+2, w          ;
        addwf   product +2, f           ;

MV_SkipSumm:

        bcf     status, C               ; Увеличиваем вес слагаемого
        rlf     work_data, f            ;
        rlf     work_data+1, f          ;
        rlf     work_data+2, f          ;

        bcf     status, C               ; Берем следующий разряд
        rrf     multiplier+1, f         ;
        rrf     multiplier, f           ;

        movf    multiplier, w           ; Прекращаем вычисления, когда заканчиваются
        iorwf   multiplier+1, w         ; биты во втором множителе
        btfss   status, Z               ;
        goto    MV_Loop                 ;
        ;------------------------------------------------------------------

        movf    product, w              ; Копируем произведение в рабочую переменную
        movwf   work_data               ;
        movf    product+1, w            ;
        movwf   work_data+1             ;
        movf    product+2, w            ;
        movwf   work_data+2             ;

        return          ; MultiplyVRef
```

```
;*****************************************************************************
;       DivideShift
;-----------------------------------------------------------------------------
; Функция выполняет деление work_data на(1<<WREG) по правилам целочисленного деления:
;
; X/Y -> (X + Y/2) / Y
;
; Использует перемнные:
;       divider   - счетчик цикла сдвигов
; На входе:
;       work_data - делимое
;       WREG      - степень делителя (1 << WREG)
; На выходе:
;       work_data
;
;-----------------------------------------------------------------------------
DivideShift:
        movwf   divider
DS_Loop:
        bcf     status, C               ; Выполняем нужное количество сдвигов,
        rrf     work_data+2, f          ; т.е. эквивалент деления на (1 << divider)
        rrf     work_data+1, f
        rrf     work_data+0, f
        decfsz  divider, f
        goto    DS_Loop

        ;---------------------------------------------------------------------
        ; На данный момент CARRY=1, если остаток больше половины делителя
        ; в этом случае нужно прибавить к частному "1"
        ;---------------------------------------------------------------------

        btfsc   status, C
        incfsz  work_data+0, f
        return
        incfsz  work_data+1, f
        return
        incf    work_data+2, f
        return

        ; DivideShift




;*****************************************************************************
;       Bin2Dec
;-----------------------------------------------------------------------------
; Функция переводит переменную work_data в массив комбинаций для вывода на
; 7-сегментный индиктор
;
; На выходе:
;       output_text
;-----------------------------------------------------------------------------
Bin2Dec:

        movlw   .100
        call    DivWreg                 ; wreg = сотни, work_data = work_data % 100
        call    GetSegments             ; Получаем соответсвующую комбинацию сегментов
        addlw   SEG_H                   ; У первого разряда зажигаем точку
        movwf   output_text + 0

        movlw   .10
        call    DivWreg                 ; wreg = десятки, work_data = work_data % 10
        call    GetSegments             ; Получаем соответсвующую комбинацию сегментов
        movwf   output_text + 1

        movf    work_data, w            ; wreg = единицы
        call    GetSegments             ; Получаем соответсвующую комбинацию сегментов
        movwf   output_text+2

        return                  ; Bin2Dec
```

```
;***************************************************************************
;       DivWreg
;---------------------------------------------------------------------------
; Деление двухбайтового числа на wreg методом вычитания
; Использует переменные:
;       div_result
; На входе:
;       work_data - делимое
;       wreg      - делитель
; На выходе:
;       wreg      - частное
;       work_data - остаток
;---------------------------------------------------------------------------
DivWreg:
        clrf    div_result
DW_Loop:
        incf    div_result, f           ; Увеличиваем счетчик вычитаний
        subwf   work_data, f            ; Производим вычитание делителя из делимого
        btfss   status, C               ;
        decf    work_data+1, f          ;

        btfss   work_data+1, 7          ; Проверка на отрицательный результат (бит 15=1)
        goto    DW_Loop
        ;-------------------------------------------------------------------

        addwf   work_data, f            ; Восстанавливаем остаток
        incf    work_data+1, f          ;

        decf    div_result, w           ; Возвращаем частное

        return


;***************************************************************************
;       GetSegments
;---------------------------------------------------------------------------
; Возвращает комбинацию сегментов 7-сегментного индикатора для конкретной цифры
; На входе:
;       wreg - чисо от 0 до 9
; На выходе:
;       wreg - комбинация сегментов
;---------------------------------------------------------------------------
        ORG     0x0400
GetSegments:
        clrf    pclath
        bsf     pclath, 2
        andlw   0xF
        addwf   pcl, f

        retlw   SEG_A + SEG_B + SEG_C + SEG_D + SEG_E + SEG_F           ; zero
        retlw   SEG_B + SEG_C                                           ; one
        retlw   SEG_A + SEG_B + SEG_D + SEG_E + SEG_G                   ; two
        retlw   SEG_A + SEG_B + SEG_C + SEG_D + SEG_G                   ; three
        retlw   SEG_B + SEG_C + SEG_F + SEG_G                           ; four
        retlw   SEG_A + SEG_C + SEG_D + SEG_F + SEG_G                   ; five
        retlw   SEG_A + SEG_C + SEG_D + SEG_E + SEG_F + SEG_G           ; six
        retlw   SEG_A + SEG_B + SEG_C                                   ; seven
        retlw   SEG_A + SEG_B + SEG_C + SEG_D + SEG_E + SEG_F + SEG_G   ; eight
        retlw   SEG_A + SEG_B + SEG_C + SEG_D + SEG_F + SEG_G           ; nine

        ; Дополняем таблицу до 16 значений, чтобы предотвратить
        ; улет неизвестно куда при неправильном значении wreg на входе
        retlw   0
        retlw   0
        retlw   0
        retlw   0
        retlw   0
        retlw   0

;***************************************************************************
;
; Конец программы Voltmetr.asm
;
;***************************************************************************

        END
```