В. Тимофеев
osa@pic24.ru

Как писать программы без ошибок.

(практическое руководство)

1. Введение

1.1. О чем это пособие

Про ошибки

По профессии я занимаюсь поиском и исправлением ошибок в чужих программах. За время работы я набрал некоторую коллекцию всевозможных багов и попытался свести их в одну таблицу и классифицировать с тем, чтобы оформить некую брошюру по быстрому поиску и исправлению наиболее часто встречающихся ошибок. Однако произвести такую классификацию не удалось. Дело в том, что, выделив 5-6 основных ошибок, таких как: неправильное приведение типов, путаница со знаковыми и беззнаковыми выражениями, пренебрежение выполнением проверок аргументов функций и пр., - я увидел, что остальные ошибки слишком индивидуальны, чтобы их как-то обобщать. О приемах поиска и говорить нечего, т.к. их намного больше, чем самих ошибок.

Тем не менее, каждый раз, исправляя какую-либо ошибку, я для себя отмечал, что ее можно было и не совершить, если бы то-то, то-то. Со временем я уяснил, что большая часть ошибок совершается из-за неправильного подхода к процессу программирования. Поэтому идея написания брошюры «Локализация и исправление ошибок» перешла в идею написания брошюры «Как не совершать ошибки». И я считаю это правильным, потому что лучше учиться строить, а не восстанавливать плохо построенное. Большая часть советов, описываемых мной, довольно банальны и просты. Вроде как, все их знают, но почему-то многие не придерживаются.

О каких ошибках идет речь. Конечно же, не о тех, которые обнаруживает компилятор при трансляции программы. Пока мы не исправим эти ошибки, нам не удастся получить код, который мы сможем прошить в контроллер. Эти ошибки в большинстве своем означают, что текст не соответствует правилам языка программирования, и, по сути, являются помарками. Также в этом материале не хотелось бы касаться так называемых «запланированных» ошибок (обработка неправильных данных, пришедших из вне, ошибочные действия пользователя и пр.). Мы будем говорить об ошибках реализации, т. е. о тех, которые будут проявляться в ходе выполнения программы. Эти ошибки являются следствием неправильно запрограммированного (или составленного) алгоритма, допуска условностей, невнимательности и неаккуратности.

Для кого это пособие

В этом пособии я изложил свой подход к программированию, привел свои правила, которыми руководствуюсь при написании программ, и обозначил те моменты, которым при написании программ уделяю особое внимание. Уверен, что моя точка зрения не единственная, а также, что где-то мне самому еще есть чему поучиться. Так что я буду рад, не только если пособие окажется кому-то полезным, но также если кто-то решит меня в чем-то поправить или дополнить.

Статья рассчитана на широкий круг программистов, поэтому я старался не затрагивать концепцию доказательного программирования, и вообще старался привести больше практических советов, чем теории (с которой сам на «Вы») и формалистики. Так же в этом пособии нет описания приемов повышения надежности программ (тестирование ROM, RAM, EEPROM, троирование, перепроверка действий и пр.).

Не следует считать, что после прочтения данного пособия ваши программы автоматически станут безошибочными, и уж тем более, что в будущем ошибок не будет вовсе. Ошибки все равно будут, но процент их сильно сократится. Более того, на написание программ по правилам, изложенным в этом пособии, будет затрачено время в два, в три, в пять раз больше, чем без правил. Писать программы без ошибок - это труд, причем кропотливый. Просто в отличие от кропотливого труда по поиску и исправлению ошибок, этот вид труда поддается планированию и формализации, т.е. является предсказуемым. А применение описанных здесь правил в разы сократит время непредсказуемого процесса отладки, который, как показывает практика, зачастую отнимает намного больше времени, чем само кодирование.

Предполагается также, что читатель знает, что такое контроллер, как он устроен, как включается, наконец, что такое электрический ток, потому что без этих знаний

заниматься микроконтроллерами бесполезно. Так что, как говорилось в старом анекдоте: «Учите матчасть!»

1.2. *«Учите матчасть!»*

Это самая очевидная рекомендация. Про нее можно было и не говорить, но для полноты картины вскользь коснемся этой темы. Программист, пишущий для контроллеров должен знать схемотехнику, устройство самого контроллера, язык программирования, на котором пишет программу, и особенности используемого компилятора. Тот программист, который пренебрегает необходимостью иметь эти знания, теряет право жаловаться на то, что его программа не работает.

Схемотехника

Как минимум нужно знать:

- основы электроники (и цифровой и аналоговой)
- законы Ома и Кирхгофа
- типовые схематические решения (RC-цепочки, транзисторные ключи)

Кроме того, нужно знать, что необходимо устанавливать диоды параллельно катушкам реле, что светодиоды включаются через токоограничивающий резистор, что у механических контактов есть такое явление как дребезг, и пр.

Контроллер

Нужно знать архитектуру контроллера, способы адресации, регистры и их назначения, периферийные модули и режимы их работы, диапазоны рабочих температур, частот, питаний и пр., нагрузочная способность портов и т.д.

Не нужно стесняться заглядывать в фирменные даташиты, там есть ответы на большинство вопросов. А то у кого-то не работает PORTA (в то время как не отключены аналоговые цепи), у кого-то не появляется «1» на выходе RA4, у кого-то прерывание по RB0 срабатывает только в половине случаев и т.д.

Язык

Конечно же, нужно знать сам применяемый язык программирования.

Для ассемблера это:

- формат команд;
- типы и количество операндов

Для языков высокого уровня это:

- семантика операторов;
- квалификаторы данных и функций;
- типы данных;
- преобразование типов;
- приоритеты операций;
- указатели (или указатели на указатели);

Компилятор

Разработчики компиляторов для микроконтроллеров, стремясь адаптировать компилятор под конкретную платформу, позволяют себе отходить от стандартов языка, одновременно не нарушая их. Каждый компилятор имеет свои особенности, незнание которых может привести к трудностям при портировании или при использовании чужих библиотек:

- набор директив;

- типы данных (размерность, знаковость);
- квалификаторы (near, far, const, rom);
- организация прерываний.

1.3. Этапы программирования

Когда ТЗ согласовано и задача формализована (переведена с проблемно ориентированных требований в технические: входные/выходные данные, режимы работы и пр.), начинается сам процесс программирования.

1. **Планирование** (включает в себя проектирование, составление плана действий, выявление требуемых ресурсов).
2. **Кодирование** (запись самой программы в машинном языке)
3. **Отладка** (локализация и устранение ошибок)
4. **Тестирование** (проверка работоспособности и соответствие спецификации)
5. Оформление проектной документации
6. Эксплуатация и сопровождение

Беда в том, что многие пренебрегают планированием, а также считают, что отладка появляется только в том случае, если была допущена какая-то ошибка при кодировании, а так как надеются с первого раза написать без ошибок, то и процесс отладки не учитывается. Однако же, в реальности отладка – это обязательный процесс, на который, вдобавок, приходится большая часть времени, сил и нервов. Закодировать даже самый сложный и запутанный алгоритм – это нетрудно, трудно заставить его работать, или, если они был написан правильно, убедиться в его работоспособности. Поэтому процессы отладки и тестирования самые длительные и самые трудоемкие. Однако, время отладки можно сократить, сделав процесс более предсказуемым и управляемым, а именно – спланировав программу.

2. Планирование программы.

Почему-то часто планированием программы вообще пренебрегают. И в лучшем случае все планирование состоит из подсчета количества требуемых выводов входа/выхода. Напомню одну старую шутку: «Нетщательно спланированная работа отнимает в 3 раза больше предполагаемого времени, тщательно спланированная – только в два». Ее можно дополнить: «Неспланированная работа отнимает все время».

Кто-то может сказать: «Я пишу маленькие программы, что там планировать?». Тем не менее, практика показывает, что лучше для маленькой программы потратить 10-15 минут на планирование (просто расписать на листе бумаги), чем тратить 3-4 дня на поиск ошибки, рытье Интернета и отбивание от обвинений в ламерстве на форумах в сети.

Ниже приведены этапы планирования программы:

1. Расписать алгоритм
2. Продумать модули
3. Продумать данные
4. Разделить периферию контроллера между процессами
5. Учесть физические свойства обвеса
6. Предусмотреть возможность расширения
7. Предусмотреть смену платформы или компилятора

Рассмотрим каждый этап более подробно.

2.1. Расписать алгоритм

Алгоритм – это ДНК программы, вернее – ее генетический код. Если в нем ошибка, то программа будет вести себя неправильно. Часто еще при составлении алгоритма на бумаге всплывают некоторые ответвления, которые могут быть проигнорированы при написании программы «в лоб». Преимущество в том, что алгоритм составляется в абстрактных терминах, еще нет ни типов данных, ни переменных, поэтому есть возможность сосредоточиться на конкретных алгоритмических моментах, не думая пока о деталях реализации. В большинстве случаев требуется составить несколько алгоритмов: один общий для всей программы, описывающий режимы работы и порядок переключения между ними, и алгоритмы работы каждого режима в отдельности. Степень детальности алгоритма, конечно, зависит от сложности программы в целом и каждого узла в отдельности.

Алгоритм может быть представлен как в виде блок-схемы, так и в виде графа переходов, в виде графика эпюр сигналов и т.д. Не стоит забывать о требованиях к проектной документации; например, если в документации требуется привести алгоритм в виде блок-схемы, то не нужно его сначала рисовать в виде графа переходов, а затем переводить в блок-схему.

2.2. Продумать модули

Нам нужно заранее продумать, из каких модулей будет собираться наша программа, вернее, на какие модули она будет разбита. Преимущества модульности я поясню ниже, а здесь приведу рекомендации, как правильно разбить программу на модули.

Во-первых, разбиение должно быть произведено по функциональному признаку. Т.е. не нужно в один модуль заталкивать USART и вывод звука на пьезодинамик, даже если при конкретной реализации это и кажется целесообразным. Дело в том, что однажды написанный модуль можно будет переносить в другие проекты, избавляя себя от необходимости писать один и тот же код по много раз. Чем модуль будет функционально изолированнее, тем проще будет его перенос. Однако и здесь не следует бросаться в крайности и делать отдельно модули для передачи данных по USART, для приема данных по USART, для подсчета и сравнения контрольной суммы данных, принятых по USART, и т.п. Функционально модуль должен быть полным, но безысбыточным.

Также стоит отметить, что при разбиении своей будущей программы на модули нужно учитывать наличие уже готовых модулей (либо своих, либо чужих).

Большинство модулей можно разделить на два типа: системные (работают на уровне сигналов и железа) и алгоритмические (работают на уровне обработки данных и режимов). Хороший пример – работа ЖКИ. Предположим, нам требуется выводить информацию на ЖКИ HD44780. Крайне неудачным решением будет такое, при котором функции вывода конкретных данных на экран, вызываемые из головной программы, и функции работы с самим HD44780, вызываемые из функции вывода данных, будут помещены в один модуль. Тут получается, что модули обоих типов – алгоритмический и системный – смешаны в один, что сильно затруднит в дальнейшем, например, использование индикатора другого типа. Если же мы четко разделим системный функционал и алгоритмический, то в дальнейшем замена типа индикатора выльется для нас всего лишь в замену системного модуля.

2.3. Продумать данные

Также на этапе планирования мы определяем для себя, какими данными будет оперировать наша программа. Причем нужно обозначить не только назначение данных и требуемый их объем, но также заранее предусмотреть их размещение (ROM или RAM, конкретный банк RAM) и область видимости (например, нам нет смысла делать видимым во всей программе буфер выходных данных i2c).

2.4. Разделить периферию контроллера между процессами

Периферийные модули контроллера помогают упростить некоторые программные узлы, а иногда просто делают их возможными (например, без АЦП мы не сможем измерить аналоговый сигнал). Но зачастую бывает, что программных узлов, требующих использование встроенной периферии, больше, чем имеется на борту у контроллера. Поэтому периферийные модули приходится разделять между несколькими задачами (типичный пример – таймеры). При проектировании программы нужно заранее распределить периферийные модули между задачами, что поможет выбрать оптимальные параметры для каждого модуля.

(Видел программу, в которой неправильно распределенные ресурсы привели к тому, что пришлось мультиплексировать управление GSM-модулем и GPS-приемником на один USART. Программа начала сбоить и, насколько я знаю, ее так и не привели в рабочее состояние.)

2.5. Учесть физические свойства обвеса

Контроллер с нашей программой будет жить не сам по себе, его будут окружать какие-то внешние схематические узлы, специализированные микросхемы и пр. Проектируя программу, нужно заранее учесть особенности всех подключенных к контроллеру устройств, цепей и микросхем.

Например, если в устройстве будут кнопки, то нужно помнить, что механические контакты дребезжат и шуршат. Это сразу должно наталкивать на использование каких-то дополнительных счетчиков и/или таймеров для подавления этих эффектов.

Также стоит помнить про «холодный» старт внешней периферии. Например, у нас есть ЖКИ, которому после подачи питания требуется 10 мс для самоинициализации, во время которой он не будет слышать команд из вне. Если этого не учесть, то может получиться так, что наш контроллер начинает инициализацию ЖКИ примерно в то же время, когда ЖКИ заканчивает свою внутреннюю инициализацию. В результате ЖКИ то будет успевать самоинициализироваться и начинать прием наших данных, то не будет. И когда мы поймем, в чем дело, и увеличим задержку перед инициализацией до 20 мс, мы с удивлением обнаружим, что теперь при включении питания, например, успевает туда-сюда сработать реле, т.к. с новой задержкой неинициализированное состояние управляющего им вывода держится достаточно для того, чтобы ток в катушке реле успел вырасти и привести к срабатыванию.

Если в нашем устройстве предполагается прием данных по радио, то мы должны учесть, что в радиоканале есть помехи, и что прием такого сигнала нельзя делать «в лоб» по фронтам, а нужно, как и в случае с обработкой сигналов от механических контактов, заводить дополнительные счетчики, таймеры и т.п.

Примеров про обвес можно придумать еще много: состязания на шине, емкостные нагрузки, внешние источники прерываний и т.д. Все это нужно учитывать еще до начала написания текста программы, заранее предусматривая порядок действий и резервируя дополнительные ресурсы (память и скорость).

2.6. Предусмотреть возможность расширения

При проектировании программы следует заранее убедиться, что в случае значительного разрастания в дальнейшем функционала (а значит и кода) будет возможность заменить выбранный контроллер более мощным с минимальными затратами. Если мы, например, решили использовать в нашей разработке контроллер из линейки PIC16, а на этапе планирования мы прикинули, что подойдет только 16F877, или 16F946, или 16F887 (короче говоря – с максимальным объемом памяти), значит, мы неправильно выбрали линейку. В этом случае нужно брать PIC18, потому что велика вероятность того, что программа в выбранный контроллер просто не влезет.

Часто встречаю на форумах крики души: «помогите оптимизировать программу, а то она не лезет в PIC18F67J60!» (прим.: контроллер из линейки PIC18, имеющий максимально возможный объем ROM = 128Кб). Это результат непродуманного выбора контроллера на этапе планирования, если этот этап вообще был проведен.

Так же надо учесть, что при отладке программы нам потребуются кое-какие ресурсы (об этом речь пойдет ниже).

2.7. Предусмотреть смену платформы или компилятора

Здесь речь, конечно, не о том, что нужно напичкать программу директивами условной компиляции, позволяющими максимально эффективно использовать возможности каждого компилятора, на который, возможно, придется переносить программу. Речь как раз о том, чтобы:

- **минимально использовать уникальные особенности конкретного компилятора**, а те куски кода, которые все-таки так пишутся, блокировать условными директивами (бывает, что компилятор простую операцию развернет черт те во что, а – кровь из носа - хочется сделать кратко и красиво);
- **не использовать недокументированные особенности компилятора**; некоторые операции могут быть реализованы различным способом, например, операция сдвига влево может после своего выполнения оставить флаг переноса, а может и изменит его, либо, выполнив оптимизацию, заменит инструкцию сдвига чем-нибудь, либо в качестве результата возьмет один из промежуточных результатов, получившихся при вычислении предыдущего выражения. В общем, нет гарантии, что флаг переноса будет установлен правильно.
- **переопределять типы данных**. Стоит учитывать, что знаковость и размерность в стандарте языка определены для каждого типа довольно расплывчато (например, в HT-PICC18 тип char по умолчанию беззнаковый, в то время как в MPLAB C18 – знаковый; или в CCS тип int – 8-битный беззнаковый, а в остальных компиляторах – 16-битный знаковый). Поэтому в каждом проекте хорошо бы иметь h-файл с переопределениями типов

```
#if defined(__PICC__) || defined(__PICC18__)
typedef signed   char    S08;
typedef signed   int     S16;
typedef signed   long    S32;

typedef unsigned char    U08;
typedef unsigned int     U16;
typedef unsigned long    U32;

typedef U08              BOOL;

typedef signed   char    S_ALU_INT;
typedef unsigned char    U_ALU_INT;
#endif
```

Обратим внимание на два типа: S_ALU_INT и U_ALU_INT – это знаковое и беззнаковое целые, имеющие размерность машинного слова для конкретного контроллера. Т.к. операции над операндами, имеющими размерность шины данных, производятся наиболее оптимально, иногда есть смысл пользоваться этими типами данных.

Примечание: далее в тексте для наглядности будут использоваться стандартные типы: char (8 бит), int (16 бит), long (32 бита).

3. Написание программы

При написании самого текста программы нужно руководствоваться двумя сводами правил:

- правила кодирования
- правила оформления.

Сами правила могут у каждого быть свои, но они должны по возможности исключать разночтения.

3.1. Кодирование

- Соблюдать модульность
- Избегать условностей
- Не делать длинных и сложных выражений
- Операторные скобки
- Операторы break и continue во вложенных циклах
- Точность вещественных чисел
- Целочисленное деление
- Правила для констант
    - Не использовать числовые константы
    - Задавать константам осмысленные значения
    - Соблюдать систему счисления
    - Заключать константы и операнды макросов в круглые скобки
    - Заключать тела макросов в фигурные скобки
- Правила для функций
    - Объявлять прототипы для всех функций
    - Проверять входные аргументы функций на правильность
    - Возвращать функцией код ошибки
    - Не делать очень больших функций
- Использовать сторожевой таймер
- Атомарный доступ

3.1.1. Соблюдать модульность

Мы уже говорили о разбиении программы на модули, еще раз приведу преимущества модульности:

- Мобильность (легкий перенос модуля в другую программу)
- Наглядность (легкий поиск определений конкретных функций)
- Заменяемость (замена одного модуля другим при изменении условий работы, например, при смене внешнего оборудования)

Раз мы решили разбить программу на модули, то нужно следовать этому решению до конца и не совершать каких-либо действий, нарушающих модульность. Т.е. наши модули должны обладать следующими характеристиками:

- **Самобытность.** Функции и переменные, относящиеся к одному модулю, определять именно внутри этого модуля. Понятно, что иначе перенос модуля в другую программу обернется тем, что в каждой новой программе придется переопределять недовнесенные переменные.

- **Самодостаточность.** Не использовать в модуле внешние переменные из верхних модулей. Опять же, причина в том, что при переносе модуля в другую программу придется в новой программе не только доопределять какие-то переменные, но и восстанавливать механизмы их работы с тем, чтобы модуль вел себя правильно.
- **Гибкость в настройке.** Например, если это модуль для работы по шине i2c, то должна быть возможность выбирать адрес устройства, разрядность адреса данных, выводы, к которым подключено устройство i2c.
- **Для каждой программы – своя копия модуля**

  Лично я считаю, что не стоит держать все модули в одной папке для всех проектов. В папку каждого проекта нужно помещать копию модуля и работать с ней. Во-первых, это позволит применить правило гибкой настройки, а во-вторых, избавит от проблем при модификации модуля. Вы сделаете малейшую модификацию модуля для нового проекта, а год назад написанная программа перестанет работать. Каждая программа должна жить своей жизнью, тем более что сейчас со свободным местом на винчестере проблем нет.

  Но тут есть и недостаток: при обнаружении ошибки в модуле ее придется исправлять во всех копиях этого модуля в разных проектах, что затруднительно. Хотя лично я считаю, что этим недостатком можно пренебречь, поскольку ошибки в уже готовых модулях обнаруживаются сравнительно редко. На мой взгляд, программа должна быть единым целым, т.к. при этом она живет своей жизнью, есть возможность ее целиком скопировать в архив, есть уверенность, что если вдруг проявился сбой, то он вызван только модификациями конкретного проекта, а не чем-то из вне.

3.1.2. Избегать условностей

В программе следует избегать любых конструкций, которые при прочтении могут быть истолкованы двояко, или которые смогут при определенных условиях повести себя неправильно. Самые частые допуски, которые позволяют себе программисты, - это несоблюдение границ использованных типов, путаница со знаковыми и беззнаковыми переменными, пренебрежение приведением типов. Эти допуски рассмотрим в первую очередь.

Типы данных.

Нужно определять переменные тем типом, который соответствует их назначению. Например, не следует переменную, которая будет использована как числовая, определять типом char. Число может быть либо signed char либо unsigned char. Сам же char определяет символьную переменную. Понятно, что для контроллера все эти переменные – одно и то же, но для компилятора, а также для человека, читающего текст программы, - это разные вещи.

Например, типичная ошибка:

```
char Counter;
Counter = 10;
if (Counter >= 0) ...;
```

Это выражение будет правильно обрабатываться в MPLAB C18, в то время как в HT-PICC18 оно всегда будет возвращать true. Все из-за того, что в стандарте языка не оговаривается знаковость типа char, и каждый разработчик компиляторов вправе толковать его по-своему. В приведенном примере переменная должна быть определена так:

```
signed char  Counter;
Counter = 10;
if (Counter >= 0) ...;
```

Приведение типов.

Не стоит надеяться, что компилятор всегда сам сделает приведение типов в случае, когда в выражении участвуют разнотипные переменные и константы. Всегда следует выполнять приведение типов вручную, исключая разночтения выражения программистом и компилятором.

Неправильно:

```
int   i;
void *p;
p = &i;
```

Правильно:

```
p = (void*)&i;
```

В конкретном примере мы, скорее всего, получим правильный результат и без приведения типов (хотя, возможны нюансы с однобайтовыми near-указателями для PIC18). Но бывает так, что программист, рассчитывая на автоматическое приведение типов, проморгает его отсутствие в более сложном выражении, например: выражение содержит подвыражения в скобках, в которых типы будут приведены только после выполнения подвыражения, т.е. тогда, когда, например, значимость будет уже потеряна (из-за переполнения).

То же самое относится к передаче параметров в функцию.

Побайтовое обращение к многобайтовой переменной.

Пример неправильного обращения:

```
unsigned char lo, hi;
unsigned int  ui;
...
lo = *((unsigned char*)&ui + 0);
hi = *((unsigned char*)&ui + 1);
```

Дело в том, что стандартом языка не предусматривается порядок чередования байтов в многобайтовых объектах.

Правильное обращение:

```
lo = ui & 0xFF;
hi = ui >> 8;
```

Определение функций.

При определении функции следует указывать полностью входные и выходные типы. Если функция определена просто:

```
myfunc ()
```

, то компилятор по умолчанию будет считать, что она возвращает int, и не принимает параметров. Однако не следует оставлять такой неопределенности.

```
    myfunc ()       // неправильно
    myfunc (void)   // неправильно
int myfunc ()       // неправильно
int myfunc (void)   // Правильно
```

Пустые операторы.

Не следует в теле цикла while, а также в операторах if…else использовать пустой оператор:

```
while (!TRMT);          // Ожидаем освобождения буфера
TXREG = Data;
```

Нечаянно пропущенная ‘;’ обернется неправильным поведением программы. Лучше вместо пустого оператора ставить `continue` либо `{}`:

```
while (!TRMT) continue; // Ожидаем освобождения буфера
TXREG = Data;
```

Про оператор switch.

В операторе **switch** нужно:

- Определять ветку **default**
- В каждом case ставить **break**
- неиспользуемые **break** закрывать комментариями

```
switch (...)
{
    case 0x00:
         ...
         break;
    case 0x01:
         ...
         // break; // Поставив закомментированный
                   // break, мы даем себе понять,
                   // что после обработки условия
                   // 0x01 мы хотим перейти к коду
                   // обработки условия 0x02, а
                   // не пропустили break случайно
    case 0x02:
         ...
         break;

    default:       // Обязательная ветка, даже
         ...       // если есть уверенность, что
                   // выражение в switch всегда
                   // принимает одно из описанных
}                  // значений
```

Неинициализированные переменные.

Нельзя пользоваться неинициализированными переменными в расчете на то, что компилятор сам сгенерит код их инициализации (например, обнулит после сброса).

Скобки в сложных выражениях.

В некоторых случаях в сложных выражениях есть смысл расставлять скобки даже тогда, когда есть уверенность в правильности приоритетов операций. Такие выражения легче анализировать, т.к. ошибка может быть не только в приоритетности операций, но и в самом выражении. Также это снижает вероятность внесения ошибки при модификации выражения.

«Такая ситуация никогда не случится!»

На эту тему можно вообще долго говорить. Вот интересный фрагмент определения типа из одной из присланных мне программ:

```
typedef struct
{
    unsigned int   seconds : 6;
    unsigned int   minutes : 6;
    unsigned int   hours   : 4;  // Неправильно задана размерность
} T_CLOCK;
```

Обратите внимание на размерность полей данной структуры. Под минуты и под секунды выделено по 6 бит, чтобы охватить весь интервал 0..59. А под часы вместо 5 бит, выделено 4. Программист, написавший это, предположил, что т.к. программа будет работать только с 8 утра, то проще выделить 4 бита (покрывая интервал в оставшиеся 16 часов) с тем, чтобы вся структура влезла в два байта, а в самой программе к значению hours всегда прибавлять 8. Надо ли говорить, что сбой не заставил себя должго ждать?

Так что не стоит забывать о первом законе Мерфи: «Если неприятность может случиться, - она случается».

Мертвые циклы.

Часто в программах встречаются участки кода, которые потенциально могут привести к зависанию. Самый распространенный пример – ожидание готовности или подтверждения при работе с внешней периферией:

Неправильно:

```
void lcd_wait_ready (void)
{
    while (!PIN_LCD_READY) continue;
}
```

Если произошла непредвиденная ситуация (обрыв линии, короткое замыкание, конденсат и пр.), то из этого цикла мы никогда не выйдем. (Разве что только WDT сработает). Поэтому нужно всегда предусматривать аварийный выход из таких циклов. Можно это делать с помощью таймера.

Правильно:

```
char  lcd_wait_ready (void)
{
    TMR0   = -100;              // Готовим таймер для фиксации таймаута
    TMR0IF =    0;

    while (!PIN_LCD_READY)
    {
        if (TMR0IF) return 0;   // Выходим с кодом ошибки
    }

    return 1;                   // Выходим OK
}
```

Правда, целый таймер выделять для этого иногда накладно, и есть смысл работать с глобальной переменной, которая будет уменьшаться в прерывании по таймеру:

```
    // Фрагмент обработчика прерывания

    if (TMR0IF && TMR0IE)
    {
        TMR0IF = 0;
        TMR0 -= TMR0_CONST;     // Обновляем таймер

        if (!--g_WaitTimer)     // Проверяем переполнение
            g_Timeout = 1;
        ...
    }
...
char  lcd_wait_ready (void)
{
    g_WaitTimer = 10;           // Готовим таймер для фиксации таймаута
    g_Timeout = 0;

    while (!PIN_LCD_READY)
    {
        if (g_Timeout) return 0;// Выходим с кодом ошибки
    }

    return 1;                   // Выходим OK
}
```

Кстати, заметьте, что в обработчике прерывания проверяется не только флаг конкретного прерывания (TMR0IF), но и бит разрешения (TMR0IE). Дело в том, что в PIC-контроллерах младшего и среднего семейства несколько прерываний могут обрабатываются одним обработчиком. И если у нас прерывание по TMR0 отключено (TMR0IE = 0), а в обработчик мы попали от другого источника (например RCIF), то без проверки битов xxxIE мы обработаем все отключенные прерывания, у которых на момент входа в обработчик оказался установлен флаг xxxIF.

3.1.3. Не делать длинных и сложных выражений

Такие выражения не только трудно анализировать, но их также трудно тестировать и отлаживать.

```
t = sqrt(p*r/(a+b*b+v))+sin(sqrt(b*b-4*a*c)/(1<<SHIFT_CONST));
```

Представьте, что программа делает вычисления неправильно и есть подозрения, что проблема в этом выражении. А это выражение даже в симуляторе не прогнать. Такие выражения желательно разбивать на несколько подвыражений:

```
A = p*r;
B = a + b*b + v;
C = b*b – 4*a*c;
D = (1 << SHIFT_CONST);

if (B == 0) ...;      // Ошибка: «деление на 0»
if (C < 0)  ...;      // Ошибка: «корень из отрицательного числа»

E = A/B;
If (E < 0)   ...;     // Ошибка: «корень из отрицательного числа»

t = sqrt(E) + sin(sqrt(C)/D);
```

Теперь наше выражение не только легко читается, но и легко тестируется и отлаживается, а кроме того, – еще и имеет механизм защиты от неправильных входных данных (этот механизм можно было бы оставить и с длинным выражением, но тогда некоторые части выражения пришлось бы пересчитывать дважды).

3.1.4. Операторные скобки

При написании фрагментов программ, содержащих вложенные циклы или вложенные условные операторы, желательно расставлять операторные скобки даже для однострочных блоков.

Рассмотрим пример одной часто встречающейся **ошибки**:

```
if (A == 1) if (B == 2) C = 3;
else C = 4;
```

Замысел программиста был таков: если **A** равняется 1, то при **B**, равном 2, присвоить **C** значение 3, иначе присвоить **C** значение 4. Т.е. программист считал, что в **C** будет занесено значение 4, если **A** не равно 1. Однако на самом деле компилятор видит это по-другому: **else** применяется к ближайшему **if**, а не к тому, который выровнен с ним в тексте. В нашем случае else относится к условию if (b == 2).

Правильно это условие нужно было записать так:

```
if (A == 1)
{
    if (B == 2) C = 3;
}
else C = 4;
```

3.1.5. Операторы break и continue во вложенных циклах

Часто встречающейся ошибкой является использование break или continue во вложенных циклах или операторе switch в расчете на то, что эти операторы сработают в рамках и внешнего цикла. Вот пример из реальной программы (он немного порезан для наглядности, и ошибка сразу бросается в глаза), который подсчитывал количество положительных и отрицательных единиц в массиве, причем нулевой элемент был признаком конца массива.

```
Positive = Negative = 0;
for (i = 0; i < MAX_ARRAY_SIZE; i++)
{
    switch (array[i])
    {
        case  0:
              break;        // Выйти из цикла (ошибка, т.к. выйдем
                            // только из switch)
        case  1:
              Positive++;
              break;
        case -1:
              Negative++;
              break;
    }
}
```

Программист решил использовать для проверки конструкцию switch, в которой, среди прочего, при нахождении элемента со значением 0 счет должен был прерваться. Однако, очевидно, что break в ветке case 0 выведет программу из switch'а, а не из цикла for.

Есть несколько способов решить эту проблему. Один из вариантов в данном случае – использовать дополнительный if:

```
Positive = Negative = 0;
for (i = 0; i < MAX_ARRAY_SIZE; i++)
{
    if (array[i] == 0) break;       // Выйти из цикла
    switch (array[i])
    {
        case  1:
              Positive++;
              break;
        case -1:
              Negative++;
              break;
    }
}
```

Однако он не очень эффективный, т.к. приводит к повторному вычислению адреса элемента массива при косвенной адресации. Есть и другие варианты решения проблемы. Главное – помнить, что break/continue прерывает/продолжает только текущий цикл (или switch).

3.1.6. Точность вещественных чисел

Когда появляется необходимость работы с вещественными числами с плавающей запятой, нужно внимательнее изучить теорию об их структуре (мантисса, порядок, знаки). У некоторых программистов, которые с ней не знакомы, при виде диапазона +/-1.7e38 появляется иллюзия всемогущества и универсальности этого типа данных. При этом из вида упускаются такие важные детали, как потеря значимости, нормирование, переполнение, относительная погрешность.

В одной программе я видел такой фрагмент:

```
long l;
float f;
f = 0.0;

for (l = 0; l < 100000L; l++)
{
    ...;
    f = f + 1.0;
    ...;
}
```

(Примечание: в программе был использован вещественный тип размерностью 24-бита.)

На месте троеточий делались кое-какие вычисления, одним из параметров в которых была переменная f. В этом цикле первые две трети результатов оказывались правильными, а дальше появлялись отклонения, причем, чем дальше, тем сильнее.

А происходило следующее: когда переменная f досчитывала до значения 65536 (т.е. выходила за пределы 16-разрядной мантиссы), ее экспонента (порядок) увеличивалась на единицу. При этом вес младшего разряда мантиссы оказывался равным 2. При выполнении сложения двух вещественных чисел f и 1.0 сперва производится приведение их к общей экспоненте, в результате которого 1.0 превращается в 0, т.к. при сложении в обоих числах порядок выравнивается в большую сторону, т.е. вес младшего разряда мантиссы в обоих операндах будет равен 2, при этом происходит потеря значимости, и результат сложения 65536.0 + 1.0 будет равен 65536.0.

Также стоит отметить, что в общем случае для вещественных чисел нельзя проверять деление умножением.

3.1.7. Целочисленное деление

Округление.

Часто встречается такой допуск при делении целых чисел, будто бы компилятор самостоятельно должен производить округление. Так вот: он этого не делает, это должен делать сам программист. Типичная ошибка, например, при расчете скорости USART'а:

Неправильно:

```
#define FOSC       200000000L
#define BAUDRATE      115200L
...
SPBRG = FOSC / (BAUDRATE*16) - 1;
```

При расчете на бумаге все выглядит красиво:

20000000/(115200*16) – 1 = 9.85

и округляется в большую сторону до 10. Ошибка скорости при этом составляет:

E = (10 - 9.85) / 9.85 * 100% = 1.5%

, что в допустимых пределах.

Однако, когда мы поручаем компилятору вычислить формулу в том виде, в котором мы ее привели, он округление произведет не по правилам математики (<0.5 – в меньшую сторону, >=0.5 – в большую), а просто откинет дробную часть. В результате ошибка будет уже:

E = (10 - 9) / 9 * 100% = 11%

, т.е. в 5 раз превышает допустимую.

Та же проблема всплывает при любых расчетах с целочисленными переменными и константами, где присутствует операция деления.

Для того чтобы округление производилось правильно, нужно к числителю прибавлять половину знаменателя:

Правильно:

```
SPBRG = (FOSC + BAUDRATE*8) / (BAUDRATE*16) – 1;
```

В общем случае формула “A = B / C” при использовании целых чисел в программе должна быть записана так:

```
A = (B + C / 2) / C;
```

Операцию деления на 2 (в выражении C/2) целесообразно заменять сдвигом вправо на один разряд. Обратите внимание, что деление на 2 – это то же самое целочисленное деление, но при выражение записывается C/2 , а не (C+1)/2, т.е. к числителю не прибавляется половина знаменателя. Почему? Не вдаваясь в подробности насчет различия погрешностей при делении на разные значения (в нашем примере деление на 2 и деление на C), приведу тривиальный пример: «разделить 1 на 1». Результат деления должен быть равен 1:

(1 + 1 / 2) / 1 = (1 + 0) / 1 = 1

Если мы при делении на 2 в дроби 1/2 прибавим к числителю половину знаменателя, то получится:

(1 + (1 + 2/2) / 2) / 1 = (1 + 2 / 2) / 1 = (1+1) / 1 = 2

Последовательность делений и умножений.

Дополню еще одной рекомендацией: если в выражении присутствуют операции как умножения так и деления, то формулу лучше переписать так, чтобы сперва выполнялись операции умножения, а затем деления:

Неправильно:

```
A = B / C * D;
```

Правильно:

```
A = B * D / C;
```

Причины очевидны: так как приоритет операций умножения и деления одинаков, то выражение будет вычисляться слева направо, причем округление будет выполняться после каждой операции. Таким образом, в первой формуле на D будет умножено не только отношение B/C, но еще и ошибка округления.

Рассмотрим пример: «2 / 3 * 3». Результат выражения должен быть равен 2. Сначала перепишем это выражение по правилам округления, как было описано выше (иначе 2/3 дадут результат 0, и результат всего выражения тоже будет нулевой):

(2 + 3/2) / 3 * 3 = (2 + 1) / 3 * 3 = 1 * 3 = 3

Как видим, мы получили неправильный ответ. Почему? Потому что на 3 была умножена еще и ошибка округления отношения 2/3. Ошибка была равна 1/3, и, помножив ее на 3, мы получили лишнюю 1 (проблема произойдет также и при округлении вниз). Правильно было сначала произвести умножение, а потом деление, т.е. «2*3/3»:

(2 * 3 + 3 / 2) / 3 = (6 + 1) / 3 = 2

Есть один тонкий момент – это переполнение, которое при перемножении нескольких чисел может возникнуть еще до того, как придет очередь до операторов деления. Но это можно предусмотреть на этапе разработки, взяв для вычислений числа большей разрядности.

3.1.8. Константы.

Не использовать числовые константы в оперативной части кода

Например, в программе предполагается использовать массив размерностью 25 элементов. Не следует по всему тексту программы явно указывать константу 25. Пример **неправильного** подхода:

```
char String[25];
...
for (i = 0; i <= 24; i++) String[i] = ‘ ‘;
```

Правильно в одном месте программы определить константу, а потом в тексте оперировать только ее именем:

```
#define STRING_SIZE      25
...
char String[STRING_SIZE];
...
for (i = 0; i <= STRING_SIZE - 1; i++) String[i] = ‘ ‘;
```

Дело в том, что если по каким-то причинам придется изменить размерность массива, то придется и по всему тексту программы выискивать все относящиеся к размерности фрагменты и исправлять их. Пропустив всего один такой фрагмент, можно получить редко проявляющийся но довольно разрушительный баг. (Обратите внимание на то, что применен оператор “<= 24”, а не “< 25”; я специально сделал для примера такой вариант, чтобы было понятно, что при изменении размерности массива простой поиск с заменой может не дать полноценного результата.)

Поэтому всем константам, не являющимся неоспоримыми (например, в минуте 60 секунд, в килобайте 1024 байта, при переводе в десятичную систему всегда число делим на 10 и т.д.), следует давать имена и в программе работать только с именованными константами.

Указывать тип константы.

Неправильно:

```
#define FOSC                4000000
#define MAX_INTERATIONS       10000
#define MIDDLE_TEMPERATURE       25
```

Если с определением FOSC сомнений не возникает (компилятор однозначно поймет, что это 32-битная константа), то с MAX_ITERATIONS будут проблемы. Если где-нибудь в коде встретится выражение:

```
(MAX_ITERATIONS * 10)
```

, то результатом его будет не 100000, а… -31072. Почему? Потому что по умолчанию константа будет рассматриваться как знаковое 16-разрядное целое. Результат 100000 выходит за границы 16 разрядов, поэтому он будет обрезан до 0x86A0, что соответствует -31072.

То же самое касается констант, которые должны быть вещественными. Если в программе встретится выражение:

```
(MIDDLE_TEMPERATURE / 26)
```

, то результатом будет 0, а не предполагаемые 25/26 = 0,96.

Правильно:

```
#define FOSC                4000000L
#define MAX_INTERATIONS       10000L
#define MIDDLE_TEMPERATURE       25.0
```

Присваивать константам осмысленные значения

Вот пример из реальной программы:

```
#define BAUDRATE     24     // 9600
```

В функции инициализации периферии была такая строчка:

```
SPBRG = BAUDRATE;
```

Программа должна была работать с тактовой частотой 4 МГц. А для такой тактовой частоты константа BAUDRATE должна иметь значение 25, а не 24. Таким образом, ошибка скорости составила 4%, что, в общем-то, почти на грани допустимого. В результате иногда принимались не те данные, которые ожидались. В разговоре с программистом выяснилось, что ранее программа работала по USART на скорости 19200 (константа BAUDRATE=12), но потом из-за смены внешнего оборудования пришлось изменить скорость на 9600, что программист и сделал, умножив константу на 2.

Но беда еще и в том, что даже с комментарием «9600» такое определение константы может заставлять задумываться. Ведь если при отладке выясняется, что что-то не так с приемом/передачей по USART, то эту константу придется проверять по формуле. Не говоря уже о том, что при выборе другой скорости или при смене тактовой частоты константу придется пересчитывать. И не надо забывать, что в программу может заглядывать не только автор, а для чужого человека заглядывание в подобный код равносильно написанию собственного.

Понятное дело, что в нашем примере программист просто ошибся с формулой (т.к. должен был сперва прибавить 1, потом умножить на 2, а затем отнять 1), но ошибки можно было бы избежать, если бы константа задавалась осмысленной:

```
#define FOSC     40000000L
#define BAUDRATE     9600L
```

А уже при присваивании регистру SPBRG переводить эту константу в нужный вид с учетом тактовой частоты, также заданной в виде константы.

```
SPBRG = ((FOSC + BAUDRATE*8) / (BAUDRATE*16) – 1;
```

Итак, константы лучше задавать в том виде, в котором мы привыкли их видеть, т.е.:

- время задавать в секундах, а не в тиках 65536 мкс;
- давление – в паскалях (или в барах), а не в единицах АЦП;
- Напряжение – в вольтах;
- температуру – в градусах;
- частоту – в герцах.

Это, во-первых, сделает программу более читабельной, а во-вторых, поможет избежать ошибок, которые будут являться следствием пересчетов из одной системы в другую. Правильнее один раз настроить формулу преобразования, чем каждый раз по ней же пересчитывать константы.

Два слова о проверке правильности задания констант.

Задавая константу в понятном для нас виде, хотелось бы одновременно быть уверенным в том, что ее значения не выходят за рамки возможностей контроллера. Например, BAUDRATE из нашего примера должна обеспечивать точность скорости +/- 2%. Или давление, которое мы задаем в барах, должно быть таким, чтобы при переводе его в единицы АЦП получилось значение в пределах 1023. Поэтому в программе нужно блокировать неправильно заданные константы условными директивами и сообщениями об ошибке. Например:

```
// Задание параметров

#define FOSC         4000000L
#define BAUDRATE        9600L

// Вычисление ошибки (в процентах)

#define SPBRG_CONST    ((FOSC + BAUDRATE*8) / (BAUDRATE*16) - 1)
#define REAL_BAUDRATE  ((FOSC + (SPBRG_CONST+1)*8)/((SPBRG_CONST + 1)*16))
#define BAUDRATE_ERROR (100L * ((BAUDRATE - REAL_BAUDRATE) + BUADRATE/2) / BAUDRATE)

// Проверка ошибки на диапазон -2%..+2%

#if BAUDRATE_ERROR < -2 || BAUDRATE_ERROR > 2
    #error "Неправильно задана константа BAUDRATE"
#endif
```

Выглядит сложно, но сложность компенсируется тем, что эта формула должна быть проверена и отлажена один раз.

Соблюдать систему счисления.

При задании константы нужно соблюдать ту систему счисления, которая подходит константе по сути. Бывает, делают так:

Неправильно:

```
TRISA = 21;    // RA0,RA2,RA4 - вход, RA1, RA3 - выход
TRISB = 65;    // 65 = 0x41 = 0b01000001
```

В данном случае нужно задавать константу именно в бинарном виде (хотя, надо помнить, что бинарная запись числа не предусмотрена стандартом Си; тем не менее, почти все компиляторы для PIC'ов такую запись понимают). Каково будет программисту поменять один бит в такой записи?

Правильно:

```
#define TRISA_CONST     0b00010101      // RA0,RA2,RA4 - входы
#define TRISB_CONST     0b01000001      // RB0, RB6 - входы
...
TRISA = TRISA_CONST;
TRISB = TRISB_CONST;
```

В этом случае четко видно, где единицы – там входы, нули выходы.

Также часто встречается довольно нелепая запись:

Неправильно:

```
TMR1H = 0xF0;
TMR1L = 0xD8;     // Отсчет 10000 тактов
```

Хорошо еще, если ее снабдят комментариями, хотя, как мы убедились на примере с BAURDATE, комментарий не гарантирует правильности константы. Опять же большие трудности возникают при пересчете констант. В данном случае нужно пользоваться именно макросом (тут есть тонкость для контроллеров PIC18: TMR1H – это буферный регистр, и нам важна последовательность присваивания: сначала старший, затем – младший; стандартом языка последовательность не предусмотрена).

Правильно:

```
#define TMR1_WRITE(timer)                \
    {                                    \
        TMR1H = timer >> 8;              \
        TMR1L = timer & 0xFF;            \
    }
...
TMR1_WRITE(-10000);
```

3.1.9. Заключать константы и операнды макросов в круглые скобки

Типичная ошибка при определении числовых констант в виде выражения – не использование скобок. Вот пример **неправильного** определения:

```
#define PIN_MASK_1    1 << 2
#define PIN_MASK_2    1 << 5
...
PORTB = PIN_MASK_1 + PIN_MASK_2;
```

Данное выражение развернется компилятором в такое:

```
PORTB = 1 << 2 + 1 << 5;
```

Вспомним, что приоритет операции «+» выше, чем приоритет сдвига, и получим выражение:

```
PORTB = 1 << (2 + 1) << 5 = 1 << 8
```

Т.е. совсем не то, что мы ожидали. Ошибки можно было бы избежать, взяв выражения при определении констант в скобки, т.е. **правильное** определение:

```
#define PIN_MASK_1    (1 << 2)
#define PIN_MASK_2    (1 << 5)
...
PORTB = PIN_MASK_1 + PIN_MASK_2;
```

Транслируется в:

```
PORTB = (1 << 2) + (1 << 5);
```

Из этой же области часто совершаемая ошибка – не заключение в скобки операндов макросов. Например, имеем макрос:

```
#define MUL_BY_3(a)    a * 3
```

Выглядит просто и красиво, однако, попробуем вызвать макрос так:

```
i = MUL_BY_3(4 + 1);
```

И вместо предполагаемого результата 15 получим результат 7. Дело в том, что макрос развернется в следующее выражение:

```
i = 4 + 1 * 3
```

Ошибки можно было бы избежать, взяв аргумент макроса в скобки:

```
#define MUL_BY_3(a)   (a) * 3
```

Тогда выражение примет вид:

```
i = (4 + 1) * 3;
```

3.1.10. Заключать тела макросов в фигурные скобки

Тематика схожа с предыдущей. Рассмотрим пример:

```
#define I2C_CLOCK()    NOP(); SCL = 1; NOP(); SCL = 0;
```

Т.е. выдерживаем паузу в один такт, затем формируем импульс длительностью два такта. Но такое определение может сыграть с нами злую шутку, если макрос будет использоваться внутри условного оператора:

```
if (...) I2C_CLOCK();
```

Компилятор развернет макрос следующим образом:

```
if (...) NOP(); SCL = 1; NOP(); SCL = 0;
```

Как видно, условием отрезается только один NOP(), а сам импульс все-таки формируется. Ошибки можно было бы избежать, взяв тело макроса в фигурные скобки:

```
#define I2C_CLOCK()    {NOP(); SCL = 1; NOP(); SCL = 0;}
```

Такие ошибки крайне трудно отслеживать, т.к. на вид все выглядит правильно, и ошибку будем искать перед условием, после условия, в выражении условия, но не в I2C_CLOCK(). Конечно, такой подход имеет свой недостаток. При использовании оператора else получим сообщение об ошибке, что в программе присутствует лишняя ';' (или что else не на месте):

```
if (...) I2C_CLOCK(); else return;
```

Все дело в том, что мы перед else и после '}' ставим ';', которая воспринимается компилятором как конец оператора if. Но гораздо проще следить за теми ошибками, о которых сообщает компилятор, чем за те теми, которые молча собираются, да еще маскируются в программе.

3.1.11. Функции

Объявлять прототипы для всех функций

Для любой функции, объявленной в модуле, нужно объявлять прототип в начале файла с тем, чтобы не задумываться при вызовах, где функция определена: выше вызова или ниже. Кроме того, при исследовании исходного текста программы это сразу дает общую картину: какие функции есть в модуле, какие параметры они принимают и какие значения возвращают. Вдобавок это позволяет сгруппировать наиболее значимые функции в начале файла, а все вспомогательные определить где-нибудь внизу, чтобы не мешались.

В заголовочный же файл нужно выносить прототипы только тех функций, которые будут вызываться из других модулей. При этом не забывать определять их с квалификатором extern.

Проверять входные данные на правильность

Перед выполнением каких-либо операций с данными нужно убеждаться в их правильности:

- Инициализированные указатели;
- Попадают в область допустимых значений (например, номер элемента в массиве должен быть меньше размерности массива, чтобы не произошла запись в стороннюю ячейку); строка, содержащая имя, не может состоять из цифр.
- Соответствуют реальности (например, счетчик битов в байте не может быть больше 8; температура в комнате не может быть -128 градусов Ц);

Часто область допустимых значений и соответствие реальности – это одно и то же. Однако, есть случаи, когда требуются дополнительные проверки. Например, если параметром является структура, то можно сначала определить, что значение каждого поля в отдельности попадает в область допустимых значений, а потом по совокупности проверять соответствие данных друг другу.

Возвращать код ошибки.

Желательно предусмотреть возможность любой функцией возвращать код ошибки. Например, при неправильных входных данных, при ошибке в вычислениях (деление на ноль, переполнение), при ошибке работы с внешними устройствами. Причем, учитывая,

что сама ошибка может произойти на самом нижнем уровне вложенных функций, нужно предусмотреть возможность передачи значения ошибки наверх.

Для возврата кода ошибки иногда удобно пользоваться неиспользуемой результатом области значений. Но есть функции, возвращающие результат, который может принять любое значение из множества значений используемого типа. Например, функция перемножения двух чисел, или функция чтения байта из EEPROM. Также может вызвать трудности ситуация, когда вложенные функции возвращают результаты разных типов, что может помешать сквозной передаче кода ошибки. В таких случаях есть смысл пользоваться глобальной переменной. Однако, такой вариант затруднителен при использовании сторонних библиотек.

Все коды ошибок должны быть предопределены константами. Нежелательно пользоваться для их определения конструкцией `enum` с неинициализированными значениями, т.к. после первого релиза все будут снабжены спецификацией, где ошибки будут расписаны по номерам, а в следующей версии будет добавлен еще один код ошибки (втиснут куда-то в середину списка), из-за чего половина констант изменят свое состояние. Поэтому их нужно определять либо через #define, либо через инициализированный список перечислений:

```
enum ENUM_ERROR_CODES
{
    // Математические ошибки
    ERROR_DIV_BY_ZERO = 0x100
    ERROR_OVERFLOW,
    ERROR_UNDERFLOW,
    ...
    // Ошибки работы с EEPROM
    ERROR_EEPROM_READ = 0x110
    ERROR_EEPROM_WRITE,
    ERROR_EEPROM_INIT,
    ...
}
```

Следует отметить, что коды ошибок, генерируемые функциями, - это не те ошибки, которые можно показывать пользователю. Например «ДЕЛЕНИЕ НА НОЛЬ» на экране стиральной машины будет выглядеть комично.

Не делать длинных функций.

Эта рекомендация схожа с рекомендацией «Не делать длинных выражений». Если функция получается длинной, то есть смысл разбить ее на несколько функций. Это даст нам следующие преимущества:

- упростит внутреннюю логику функции;
- сделает ее более читабельной;
- упростит тестирование и отладку.

3.1.12. Использовать сторожевой таймер

Сторожевой таймер в контроллерах имеет три применения:

- пробуждение контроллера из режима SLEEP с заданным инетервалом;
- произведение аппаратного сброса при зависании программы.
- Для PIC-контроллеров младшего семейства – выполнение программного сброса

Рассмотрим для начала второе применение, т.е. использование его в качестве дополнительного механизма защиты от зависания. Понятно, что лучше создавать правильные независающие алгоритмы и описывать их правильными независающими программами, но, к сожалению, часто даже в отлаженной и переотлаженной программе могут скрываться ошибки, которые при определенном стечении обстоятельств приведут к зависанию. Вдобавок можно сказать, что сложные программы с большим количеством состояний и параллельных процессов бывает сложно отладить и протестировать полностью. Полное тестирование таких программ может затянуться на время, превышающее экономически целесообразные сроки выпуска устройства. В таких случаях приходится идти на некий компромисс между полнотой теста и сроками

выпуска, т.е. программа может выйти, грубо говоря, недоотлаженной. Это не значит, что она будет сбоить на каждом шагу и выдавать какие-то результаты, не соответствующие спецификации. Это значит, что при определенных условиях (чаще при совокупности внештатных ситуаций) возможно неадекватное поведение программы. И здесь нас может выручить сторожевой таймер, который не даст программе зависнуть наглухо. Конечно, он не является панацеей от неправильно составленного или неправильно запрограммированного алгоритма. Он всего лишь перезапустит программу, но не исправит допущенной ошибки. Тем не менее, с ним, по крайней мере, есть возможность восстановить работоспособность устройства.

Когда нужно обрабатывать WDT.

Во многих программах вижу «интересный» прием программистов: в битах конфигурации включали WDT, а в самой программе сбрасывали его везде, где попало, лишь бы не произошло переполнение: и в главном цикле программы, и в прерывании, и в функции задержки. Сделают так – и радуются: «Ай, какую я надежную программу написал! С вочдогом!» Этот подход эквивалентен подходу без использования WDT с той разницей, что у программиста появляется еще одна иллюзия по поводу отказоустойчивости его программы.

Задача сброса WDT совсем нетривиальна, в каждой конкретной программе нужен свой алгоритм. В самом простом случае можно завести отдельную переменную, в которой каждая подпрограмма будет устанавливать свой бит, а отдельная подпрограмма обработки WDT будет проверять эту переменную, и только в том случае, когда все требуемые биты установлены, будет производиться очистка WDT. Но такой способ подойдет только для несложной программы с небольшим количеством состояний и с ограниченным временем пребывания в одной функции.

Для сложных программ требуются разработки индивидуальных алгоритмов, которые бы отслеживали не только факт выполнения каких-то функций, но также следили за тем, чтобы функции выполнялись в правильной последовательности, и чтобы время выполнения каждой функции было в пределах допустимого.

Здесь хотел бы пару слов сказать про третье применение WDT, т.е. программный сброс в контроллерах младших семейств, т.к. они не имеют отдельной инструкции, вроде RESET в PIC18. В системах с повышенными требованиями по надежности программа должна иметь механизмы контроля правильности работы. Т.е. следить за стеком, за порядком действий, временами выполнения отдельных узлов, настройками периферийных модулей и т.д. При обнаружении отклонений, которые не могут быть исправлены на лету (например, замечено, что в функцию попали не через вызов CALL, а каким-то другим путем), единственный выход – это сброс. Но если в контроллерах среднего и старшего семейств есть инструкции, выполняющие сброс, то в контроллерах младшего семейства единственный способ выполнить сброс программно – это дать WDT досчитать до конца.

Что делать, если произошел сброс по WDT.

Порядок действий при сбросе контроллера (не только по `WDT`) - это отдельная тема (довольно большая). Здесь опишу только в двух словах.

Сделать нужно две вещи:

- Восстановить работу (по возможности сделать это незаметным для пользователей)
- Выяснить причину сброса.

С первым все понятно: все переменные, отвечающие за текущий режим работы, за состояние процессов, какие-то текущие рабочие данные и пр. должны быть определены в неинициализируемой части RAM, т.е. в той, в которую код стартапа ничего не пишет при сбросе. Для HT-PICC такие переменные должны быть определены с квалификатором persistent:

```
persistent unsigned char Mode;
persistent unsigned int  LastData[10];
```

Для MPLAB C18 они должны быть определены в секции udata, т.е. не инициализироваться при определении.

В самом начале функции main нужно проверять причины сброса (биты POR, BOR, TO, PO, RI и пр.), и в зависимости от их состояния принимать решение о том, восстанавливать ли прежний режим работы, начинать ли все с нуля, переходить ли в аварийный режим и т.д.

Теперь о выяснении причин сброса. Здесь многое зависит от стадии разработки устройства и его доступности. В зависимости от того, отлаживаем ли мы устройство, тестируем ли в реальных условиях или же устройство в эксплуатации, в зависимости от доступности самого устройства разработчику, от способности самого устройства известить о несанкционированном сбросе, от средств его связи с внешним миром и пр. факторов – средства выяснения причин могут быть различными. Мы не будем говорить о технике получения данных, которые нам смогут помочь установить причину сброса, из контроллера. Это может быть что угодно: сброс данных в отдельную страницу внешней EEPROM, передача их через какой-нибудь интерфейс для протоколирования, отображение на мониторе (или ЖКИ) и т.д. Я только укажу, какие данные могут быть полезными в общем случае:

- Содержимое стека
- Значения регистров указателей
- Значение регистров PCLATH, PCLATU
- Указатель табличного чтения ROM (TBLPTR)
- Регистр EEADDR

По этим данным зачастую можно сделать вывод о том, в каком месте произошел сбой. При использовании компиляторов от microchip для получения этих данных придется писать свой код startup, т.к. фирменная функция заранее предустанавливает регистры FSR (WREG14, WREG15 для MCC30). В каждом частном случае могут понадобиться дополнительные данные: текущий режим, какие-то индикаторы обработки критических участков кода и пр. Чем больше будет данных, тем проще будет анализировать причину сброса. Но нужно также искать некий компромисс, чтобы не загромождать дамп всякой ерундой, а еще, чтобы не дать пользователю запаниковать, если устройство уже в эксплуатации.

3.1.13. Два слова об операторе GOTO

Многие авторитетные источники строго не рекомендуют использовать оператор goto при написании программ на языках высокого уровня. Основными недостатками его применения называется сильное ухудшение читабельности текста программы и нарушение ее структурности. Да и вообще неизвестно, через что мы перепрыгиваем, выполняя goto (может быть через объявление переменной). Кроме того, часто упоминается то, что формально доказано, что любая программа, использующая goto, может быть переписана без его использования с полным сохранением функциональности. Доводы весьма убедительны, а небезызвестный Дейкстра свою статью «Доводы против оператора goto» вообще начинает с тезиса «… квалификация программистов – функция, обратно зависящая от частоты появления опетаоров goto в их программах».

Сам я не люблю использовать этот оператор и испытываю большие трудности с анализом чужого кода, где он применяется. Однако, применительно к микроконтроллерам с ограниченными ресурсами я бы оправдал использование goto в некоторых случаях.

Выход из вложенных циклов.

Этот пример приводится чаще всех остальных. Действительно, использование оператора goto выглядит довольно простым (и наглядным) решением для выхода из циклов вида (удобны, например, при поиске вариантов решений в многомерных массивах):

```
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 15; j++)
    {
        if (...) goto BREAK_LOOP;
    }
}
BREAK_LOOP:
```

В принципе, при чтении такого кода не вызывает трудностей найти метку, т.к. по смыслу операции понятно, что она внизу, после закрывающей скобки верхнего цикла. Ярые противники goto приводят два варианта альтернативного кода, позволяющего избавиться от этого оператора.

Вариант 1 – переписать цикл в виде функции.

```
void loop_func (void)
{
    int i, j;
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 15; j++)
        {
            if (...) return ;
        }
    }
}
```

Вариант 2 – использовать переменную-флаг.

```
bool StopLoop = false;
for (i = 0; i < 10 && !StopLoop; i++)
{
    for (j = 0; j < 15 && !StopLoop; j++)
    {
        if (...)
        {
            StopLoop = true;
            break;
        };
    }
}
```

(В принципе есть еще альтернативы, но они контекстно-зависимые и в общем случае применимы не во всех случаях)

Оба варианта полностью работоспособны, однако имеют свои небольшие недостатки, когда речь идет о работе с микроконтроллерами, имеющими дефицит ресурсов. Первый вариант не очень удачен, поскольку требует дополнительный свободный уровень стека (помним, что в PIC16 их всего 8, а в PIC18 - 32), что иногда может оказаться критичным. Второй вариант требует использования дополнительной переменной, что также существенно для контроллеров с малым ОЗУ. Т.е., применяя goto, мы имеем возможность сэкономить ресурсы контроллера. (Об экономии скорости я здесь не говорю, поскольку на фоне цикла из 10x15=150 итераций лишний вызов/возврат или две лишних проверки флага StopLoop будут несущественны).

«Стандартные» метки.

Лично я позволяю себе использовать метки, которые могут считаться универсальными почти для любого случая. Таких меток немного. Типичный пример – выход из функции при обнаружении ошибки в ходе ее выполнении. Ошибку функция может обнаружить на любом этапе своего выполнения (как при проверке аргументов на правильность, так, например, и при работе с внешней периферией, и при проверке контрольных сумм и т.д.), а при выходе ей требуется освободить занимаемые ресурсы и установить флаг возникновения ошибки, так что обычный return нас может не устроить просто потому, что перед каждым return придется выполнять одну и ту же последовательность действий.

Главное для меня – местоположение таких меток всегда однозначно (например, понятно, что метка, куда переходит функция при обнаружении ошибки, находится в конце функции), их использование безопасно и с точки зрения экономии ресурсов целесообразно (не любой алгоритм удобно программировать приемами структурного программирования, есть случаи, когда такие приемы делают код менее читабельным и более ресурсоемким).

Оптимизация.

Некоторые операции, такие как расчеты или работа с быстрыми сигналами, требуют оптимизации кода по скорости. В таких случаях я предпочитаю использовать операторы goto вместо if…else, break, continue и пр. из соображений экономии времени выполнения, даже в ущерб наглядности. Такие участки кода должны быть тщательно проверены и перепроверены и детально откомментированы.

3.1.14. Атомарный доступ

Часто причиной ошибки в программах может быть конфликт одновременного доступа к переменным из основной программы и из прерывания (или из прерываний с разным приоритетом). Самый простой пример: на 8-битном контроллере имеем 16-битную переменную, которая обрабатывается как в основной программе, так и в обработчике прерывания. Конфликты при работе с ней могут возникнуть, если во время обращения к ней из основного тела программы возникает прерывание, обработчик которого также захочет с ней поработать. Если в одном из случаев (или в обоих) в переменную производится запись, то могут возникнуть проблемы.

```
unsigned int ADCValue;    // Результат чтения АЦП (производится в
                          // прерывании
...
// Фрагмент обработчика прерывания
If (ADIF && ADIE)
{
    ADIF     = 0;
    ADCValue = (ADRESH << 8) | ADRESL;  // Читаем последнее измерение
    ADGO     = 1;                       // Запускаем следующее
}
...
// Фрагмент основного тела программы
Data[i] = ADCValue;      // Примечание: Data[] - массив uint'ов
```

Я думаю, не нужно пояснять, что произойдет, если в момент чтения **ADCValue** в основной программе (а чтение произойдет в два этапа: сначала младший байт, затем старший), произойдет прерывание по завершению измерения ADC? В двух словах: на момент начала чтения переменная содержала значение **257** (0x101). Первым считается младший байт и скопируется в младший байт элемента массива **Data[i]**; далее произойдет прерывание, и в **ADCValue** запишется вновь измеренное значение напряжения, которое с момента последнего изменение стало, например, меньше на 3 единицы младшего разряда, т.е. **254** (0x0FE). Возвращаемся из прерывания и продолжаем копировать **ADCValue** в **Data**[i]; нам осталось скопировать старший байт, а он стал равным 0. Вот и получается, что в **Data**[i] окажется значение **0x001**.

Часто встречал у людей недопонимание методов борьбы с этим явлением. Многие программисты считают, что их спасет квалификатор **volatile**, т.е. достаточно определить переменную, к которой есть доступ и в основном теле программ и в прерывании так:

```
volatile unsigned int ADCValue;
```

- и проблема будет решена автоматически на уровне компилятора. Однако, это не так. Квалификатор **volatile** всего лишь сообщает компилятору, что не нужно производить оптимизацию кода с участием этой переменной. А это в свою очередь даст возможность программисту блокировать прерывания на время обращения к ней, но делать это надо вручную:

```
di();
Data[i] = ADCValue;
ei();
```

Более детально про атомарный доступ рекомендую почитать [6].

3.2. Оформление

- Использовать удобный инструментарий
- Давать осмысленные имена идентификаторам
- Форматировать текст по одним и тем же правилам
- Снабжать исходный текст комментариями

3.2.1. Удобный инструментарий (редактор)

Очень важная составляющая успешного программирования – инструментарий. Применительно к написанию текста программы – это в первую очередь текстовый редактор. Если он удобный, функциональный и имеет интуитивно понятный интерфейс, то форматирование будет производиться легко и непринужденно. Помимо самого ввода текста, редактор может обеспечить нам:

- автоматические отступы;
- замену символа табуляции пробелами;
- подсветку синтаксиса;
- контекстную подстановку;
- перекрестные ссылки внутри исходного текста или целого проекта;
- вызов компилятора для сборки проекта из редактора

(Рекомендую SlickEdit).

3.2.2. Именование идентификаторов

Задавая имена переменным, функциям, константам, типам, перечислениям, макросам, файлам и пр., есть смысл следовать некоторым правилам:

- Имя должно быть осмысленным (не i, а Counter)
- Имя должно быть содержательным (не Counter, а BitsCounter)

И ниже приведем кое-какие отдельные правила именования различных объектов программы.

Именование функций.

Имя функции внутри модуля, которую предполагается вызывать из других модулей, должно начинаться с префикса, обозначающего имя модуля. Зачем это нужно? Например, в одной программе вы используете память 24LC64, для которой определяете функцию `write_byte()`. Потом вы написали другую программу, которая работает с внутренней EEPROM контроллера, и для нее тоже определили функцию `write_byte()`. А потом понадобилось сделать проект, в котором будет использоваться и внешняя EEPROM, и внутренняя. Тут-то вы и столкнетесь с последствиями неправильного именования, потому что из-за одинаковых имен модули нельзя будет объединить в одной программе. Поэтому в самом начале нужно было предусмотреть префиксы имен функций (а запись байта – это операция универсальная, она есть в работе и с дисплеем, и с модемом и еще с чем угодно).

`i2c_write_byte` – для работы с памятью 24LC64;

`eeprom_write_byte` – для работы с внутренней EEPROM;

`lcd_write_byte` – для работы с LCD и т.д.

Кроме того, имя функции должно быть кратким и, по возможности, должно соответствовать системе именования: <модуль>_<действие>_<объект>[_<суффикс>] (могут быть в другом порядке, могут разделяться символом “_”), где:

<модуль>_<действие>_<объект>[_<суффикс>]

- **модуль** – имя (или сокращенное имя) модуля, в котором определена функция;
- **действие** – глагол, определяющий назначение функции (write, read, check, count и т.д.)
- **объект** – существительное, определяющее параметрическую составляющую функции (byte, checksum, string, data и пр.)
- **суффикс** – необязательное поле, отражающее какую-либо дополнительную характеристику функции (rom, timeout).

Конечно, все имена функций под одну гребенку подвести довольно трудно, и обязательно в любой системе именования будут исключения. Но не нужно называть функции так:

Неправильные имена функций:

```
CopyOneByteFromROMToRAM();
Check();
CompareAndGetMax();
```

В заключение надо сказать, что для некоторых функций можно оставить зарезервированные имена: atof(), init(), printf() и т.д.

Именование констант

- Заглавными буквами
- Слова разделяются символом ‘_’
- Префикс в виде имени модуля или функционального назначения

```
// Определения параметров шины i2c
#define I2C_DEV_ADDR         0xA0
#define I2C_ADDR_WIDTH         16
// Определения цветов RGB
#define CL_RED           0xFF0000
#define CL_YELLOW        0xFFFF00
#define CL_GREEN         0x00FF00
```

Именование типов.

- Заглавными буквами
- Слова разделяются символом ‘_’
- Префикс в виде «T_», «TYPE_» и т.п.

```
typedef struct
{
    unsigned long   seconds : 6;
    unsigned long   minutes : 6;
    unsigned long   hours   : 5;
} T_CLOCK;
```

Именование переменных или два слова о венгерской нотации.

- Слова начинаются с заглавной буквы
- Пишутся без разделителя

```
unsigned char    BytesCounter;
signed   long    XSize, YSize;
```

О «венгерской нотации»

Некоторые используют систему именования, в которой каждой переменной приписывается префикс, показывающий ее тип. Это часто может оказаться полезным при анализе чужого (а часто и собственного) кода. Например:

```
for (Counter = 0; Counter < 40000; Counter++) ...;
```

В данном примере мы не можем точно сказать, правильно ли у нас использована переменная в данном цикле или нет. Переменная Counter может быть, во-первых, 8-битной; во-вторых, она может быть знаковой (т.е. всегда будет меньше 40000). И для того, чтобы определить правильность использования переменной, нам нужно найти определение переменной в файле. Но если мы изначально предусмотрели в имени переменной ее тип, то это избавит нас от проделывания лишней работы:

```
for (wCounter = 0; wCounter < 40000; wCounter++) ...;
```

Теперь, глядя на эту запись, мы точно можем сказать, что здесь нет ошибки.

Также системой именований можно предусмотреть область видимости переменной.

Префиксы:

- Префикс области видимости

```
Без префикса – локальная или параметр функции
s_ - статическая
m_ - локальная для модуля
g_ - глобальная
i_ - Обрабатывается в прерывании
```

- Префикс типа

```
uc – unsigned char
sc – signed char
ui – unsigned int (n)
si – signed int (w)
И т.д.
```

Имя нашей переменной может выглядеть так:

```
static unsigned char s_ucRecCounter;
```

Встречая ее в любом месте функции, мы сразу понимаем, что:

- Эта переменная предназначена для подсчета принятых байтов;
- Эта переменная имеет тип unsigned char, т.е. может принимать значения 0..255;
- Эта переменная статическая, т.е. сохраняет свое значение после выхода из функции.

Примечание. Можно для себя иметь некоторый набор зарезервированных имен для переменных общего назначения, которые встречаются наиболее часто. Например: i, j – signed int; a, b – char;s – char; f - float; и т.д. Но эти имена желательно согласовывать, во-первых, с применяемыми именами в литературе, а во-вторых, внутри собственной команды разработчиков, чтобы не было так, что у одного i – signed int, а у другого i – unsigned int.*

Преимущества венгерской нотации:

- Предотвращает ошибки использования типов
- В большом коде помогает не запутаться в переменных
- Упрощает чтение чужого кода

Недостатки венгерской нотации:

- Ухудшают читабельность кода
    - Выражение из 3-4 переменных уже трудно читается

    - Префикс может получиться очень длинным
- Изменение типа переменной влечет изменение ее имени во всех файлах
- Префикс не дает гарантию правильности задания типа, из-за чего можут получиться ложная уверенность в корректности применения переменной:

```
static signed char s_ucBytesCounter;
```

3.2.3. Форматирование текста

Очевидно, что форматирование нужно для того, чтобы программа была более наглядной. Рекомендую в качестве примера ознакомиться с документом “an_2000_rus.pdf“, где описаны правила форматирования текста программ, сформулированные для сотрудников компании micrium. Не обязательно точно следовать приведенным в нем правилам, но следует посмотреть, на чем сосредоточили внимание авторы документа, и принять это на вооружение:

- Следовать одному стилю
- При работе в команде следовать всей командой одним и тем же правилам
- Не оправдывать отсутствие форматирования нехваткой времени

Я не буду пересказывать этот документ, а только вкратце приведу основные моменты:

Текст файла должен быть разбит на секции.

Когда мы берем в руки книгу, мы знаем, что у нее помимо самой содержательной части есть титульные данные, оглавление, выпускные данные, а часто еще алфавитный указатель, список литературы. Кроме того, мы знаем, что каждый элементы каждой из перечисленных групп находится в одном месте, а не разбросаны по всем страницам. Все эти данные упрощают восприятие книги: на титуле в двух словах сказано, о чем (иногда – для кого) эта книга, по оглавлению можно быстро найти интересующую главу, по содержанию – понять содержание и т.д.

Для того чтобы текст программы легче читался, он также должен быть разбит на части (или секции), каждая из которых имеет свое назначение:

- Заголовок файла (с информацией о назначении файла, авторе текста, используемом компиляторе, дате создания и пр.);
- Хронологию изменений
- Секция включаемых файлов
- Секция определения констант
- Секция определения макросов
- Секция определения типов
- Секция определения переменных (глобальные, затем локальные)
- Секция прототипов функций (глобальные, затем локальные)
- Секция описания функций.

Для секций есть свои правила:

- Каждая секция должна содержать только те описания, которые ей соответствуют
- Секции должны быть едины, а не разбросаны по всему файлу.
- Каждой секции должен предшествовать хорошо заметный блок комментария с названием секции.

Горизонтальные отступы

Горизонтальные отступы служат для визуального подчеркивания структуры программы. Я каждый блок операторов делаю с отступом на 4 пробела вправо от верхнего блока.

Вертикальное выравнивание

При определении переменных: типы – под типами, квалификаторы – под квалификаторами, имена переменных – под именами, атрибуты – под атрибутами. В самой программе: при присваивании блока переменных желательно выравнивать знаки «=».

```
static unsigned char    BytesCounter;
static          int     Timer;
                double  Price;
                char   *p, c;
...
{
    BytesCounter = 0;
    Timer        = 0;
    Price        = 1.12;
}
```

Хочу отдельно внимание обратить на то, что '*', показывающая, что переменная является указателем, ставится рядом с переменной, а не с типом. Сравните две записи:

```
char*    p, c;
и
char    *p, c;
```

Первая запись может быть ошибочно прочитана так: переменные `p` и `c` имеют тип `char*`. На самом же деле указателем будет только p. Во второй записи это наглядно отражено.

Не делать в строке больше символов, чем помещается на одном экране

Например, если функция содержит много параметров, то имеет смысл каждый параметр писать с новой строки. Длинные выражения можно также разбивать на строки.

```
int sendto ( SOCKET                 s,
             const char            *buf,
             int                    len,
             int                    flags,
             const struct sockaddr  *to,
             int                    tolen
           );
```

Обращу ваше внимание, что при определении функции в нескольких строках правилом вертикального выравнивания стоит пользоваться не в полной мере. Попробуйте переписать эту функцию, ставя квалификаторы под квалификаторами и т.д. Он станет выглядеть довольно безобразно и совершенно нечитабельно, так что в данном случае лучше все определения начинать с одного столбца.

Одна строка – одно действие

Разделять функциональные узлы или конструкции (for, if, …) пустыми строками

Пробелы между операндами и операциями

Ниже пример форматирования по трем последним правилам:

Неправильно:

```
for(i=0;i<10;i++)a[i]=0;
if(j<k){a[0]=1;a[1]=2;}else{a[0]=3;a[1]=4;}
```

Правильно:

```
for (i = 0; i < 10; i++) a[i] = 0;

if (j < k)
{
    a[0] = 1;
    a[1] = 2;
}
else
{
    a[0] = 3;
    a[1] = 4;
}
```

3.2.4. Комментирование

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»

Для кого пишутся комментарии.

Комментарий программистом пишется в первую очередь для самого программиста. За отсутствие комментариев приходится платить временем. Чаще - своим, реже – чужим.

Содержание комментариев.

Частенько в присланных мне программах вижу, что местами комментарий написан только для того, чтобы он там был. Такое ощущение, что человек знает, что комментарий написать надо, но не знает зачем.

Что должно быть в комментариях:

- Спецификация функций:
    - что делает;
    - входные и выходные параметры (типы и краткое пояснение);
- Назначение объявляемой переменной, определяемой константы, типа, макроса;
- Краткое, емкое, безызбыточное описание действия или пояснение к нему;
- Пометки об изменениях:
    - версия (или дата) и номер пометки.
    - (Причины и описание изменения желательно держать в отдельном месте, т.к., во-первых, текст описания причин изменений и производимых действий может получиться громоздким и будет засорять основной текст программы, а во-вторых, одна причина может повлечь за собой изменения нескольких участков программы.)
- Указание отладочных узлов и временных конструкций

Пример спецификации функции:

```
/************************************************************
 *
 *  Function:       rs_buf_delete
 *
 *-----------------------------------------------------------
 *
 *  description:    Удаляем N байт из буфера
 *
 *  parameters:     uchar N - количество удаляемых байтов
 *
 *  on return:      void
************************************************************/

void rs_buf_delete (uchar N)
{
    ...
```

Пример пометок об изменениях:

```
/************************************************
 * Список изменений
 * ...
 * Версия 1.6 (22.10.2009):
 *     1. ...
 *     2. ...
 *     ...
 *     8. Иванов И.И., 17.09.2009: В функции
 *        rs_buf_delete добавлена проверка
 *        входного аргумента на 0
 * ...
 ************************************************/
...
void rs_buf_delete (signed char N)
{
    // *1.6-8* if (N < 0) return;
    if (N <= 0) return;          // *1.6-8*
    ...
```

Пример указания отладочного узла:

```
void rs_buf_delete (signed char N)
{
    if (N <= 0) return;

    /*D*/  DebugCounter++;
    /*D*/  PIN_DEBUG = 1;
    /*D*/  delay_ms(5);
    /*D*/  PIN_DEBUG = 0;
    ...
```

Чего в комментариях быть не должно:

- Эмоций

```
RB0 = 1;          // Без этой хрени не работает.
```

- Описания устаревших действий

```
if (BufSize > 0) // Буфер не пуст, флаг разрешения
// вывода установлен
```

- Дублирования действия

```
BufSize = 0;      // Обнуляем переменную, показывающую
// размер буфера
```

- Бесполезной информации

```
if (N < 15)       // Лучший вариант сравнения
```

- Непонятных сокращений и жаргона:

```
A = X / Y + Z;    // В (*1*), т.н.у., обход
```

- Ложной или вводящей в заблуждение информации:

```
if (timer < 100 && V < 10) // Если за 100мс напряжение
                           // стало выше 10 вольт
```

Расположение комментариев.

Не следует мешать код и текст комментариев в одну кучу. Например, так:

Неправильный подход:

```
/* Инициализация портов*/
PIN_DATA = 0;
PIN_CLC = 0;

/* Очистка буфера */
for (i = 0; i < BUF_SIZE; i++) Buf[i] = 0;

/*Ожидание данных */
while (ReadData())
{
    ...
```

Комментарии нужно писать так, чтобы они не сливались с кодом. Один из вариантов – выносить их на поля, выравнивая по вертикали.

Правильный подход:

```
PIN_DATA = 0;                   /* Инициализация портов    */
PIN_CLC = 0;

for (i = 0; i < BUF_SIZE; i++)  /* Очистка буфера          */
    Buf[i] = 0;

while (ReadData())              /*Ожидание данных          */
{
    ...
```

Многострочные комментарии.

При написании многострочного комментария (например, описывающего функцию), нужно, чтобы каждая строка начиналась с символа, обозначающего, что это комментарий.

Неправильный подход:

```
/* Эта функция считывает начальные установки из EEPROM,
   проверяя контрольную сумму. Если контрольная сумма не
   сошлась, то будут приняты установки по умолчанию. */
   for (i = 0; i < BUF_SIZE; i++) ...;
```

Правильные подходы:

(не очень удобный вариант: из-за наличия правого ограничителя такой комментарий утомительно редактировать)

```
/* Эта функция считывает начальные установки из EEPROM,    */
/* проверяя контрольную сумму. Если контрольная сумма не   */
/* сошлась, то будут приняты установки по умолчанию.       */
   for (i = 0; i < BUF_SIZE; i++) ...;
```

(альтернативные варианты)

```
/*
 * Эта функция считывает начальные установки из EEPROM,
 * проверяя контрольную сумму. Если контрольная сумма не
 * сошлась, то будут приняты установки по умолчанию.
 */
   for (i = 0; i < BUF_SIZE; i++) ...;
```

```
// Эта функция считывает начальные установки из EEPROM,
// проверяя контрольную сумму. Если контрольная сумма не
// сошлась, то будут приняты установки по умолчанию.
   for (i = 0; i < BUF_SIZE; i++) ...;
```

Часто удобно пользоваться горизонтальными разделителями для визуального отделения логически разных блоков в тексте программы:

```
//-------------------------------------------------
```

или

```
/*************************************************/
```

Содержательная часть комментария.

Вспомним, что мы говорили про именование идентификаторов и использование именованных констант. Рассмотрим пример:

```
switch (result)           // Проверяем состояние модема
{
    case 0:               // Модем выключен
         …
         break;
    case 1:               // Модем готов к работе
         …
         break;
    case 2:               // Модем производит дозвон
         …
         break;
    …
}
```

Если бы использовали осмысленные имена (и переменной и констант), то сам фрагмент программы оказался бы понятным и без комментариев.

```
switch (ModemState)
{
    case MODEM_OFF:
         …
         break;
    case MODEM_READY:
         …
         break;
    case MODEM_CALLING:
         …
         break;
    …
}
```

Тем самым у нас появляется возможность комментарием пояснить какой-нибудь нюанс какой-то ситуации или какого-то действия.

Формулировка.

Совет: формулируя комментарий, всегда представляйте, как будто комментарий читает другой человек. Это поможет сделать формулировку более четкой.

4. Отладка и тестирование

Отладка, как и тестирование, – неотъемлемая часть разработки. Я отвел для этих понятий один общий раздел, т.к. обычно, эти этапы производятся параллельно. Надо сказать, что эти два этапа дополняют друг друга.

Рассмотрим следующие моменты:

- Инструменты
- Резерв по ресурсам
- Заглушки и тестеры
- Предупреждения при компиляции
- Вывод отладочной информации
- Возможность блокировки вывода
- Резервное копирование

4.1. Инструменты.

В арсенале программиста может быть довольно мощный инструментарий: симулятор, внутрисхемный отладчик, лог. анализатор, осциллограф и т.д. В целом, неизвестно, сколько ошибок нужно будет обнаружить и исправить, поэтому процесс отладки и тестирования может сильно затянуться. И здесь наличие хорошего инструментария и умение с ним работать поможет сэкономить массу времени и сделает сам процесс более легким. Поэтому не нужно скупиться на средства отладки.

4.2. Резерв по ресурсам

Желательно еще на этапе планирования предусмотреть возможность взять для отладки контроллер с запасом по ресурсам. Для чего они могут понадобиться? В процессе тестирования или отладки программы нам может понадобится контроль хода выполнения или каких-то внутренних переменных. Для этого нам понадобится запас:

- периферийных возможностей
- памяти для размещения отладочного кода
- резерв по скорости

Запас по периферии.

- Внутренняя периферия контроллера.

  Здесь масса вариантов. Как минимум – иметь свободные выводы контроллера для возможности устанавливать на них логические уровни, соответствующие состоянию программы. Если в устройстве не планируется использовать модуль USART, то было бы удобно оставить свободными выводы контроллера, на которые этот модуль работает. Тогда отладочную информацию можно будет сливать, используя аппаратные возможности контроллера. Если использование USART все же планируется, а отладочную информацию все равно хочется слать через RS-232 в компьютер, то можно пожертвовать системным временем и сделать функцию вывода программно на любом свободном выводе контроллера (функция ввода нужна не так часто, а функция вывода реализуется довольно просто). Зачастую есть возможность скидывать отладочные данные в общем потоке данных (по UART, по радиоканалу).

  Также для отладки и тестирования может понадобиться аппаратный таймер для определения скорости выполнения некоторых участков или времени реакции на событие. Так что, если есть возможность на этапе планирования зарезервировать один аппаратный таймер для нужд отладки, то лучше сделать это.

- Внешняя периферия.

Например, если используется LDC, то можно предусмотреть какой-то сегмент для вывода отладочной информации. В EEPROM можно выделить отдельную страницу, куда сохранять состояние контроллера после сброса, если при запуске выясняется, что сброс был аварийным (например, туда можно записать содержимое стека, пускай сам указатель и не сохранился).

Также можно на плате предусмотреть дополнительные кнопки, дополнительные светодиоды.

Память для размещения отладочного кода.

Здесь все понятно: отладочный код занимает место в ROM и часто требует каких-то ячеек RAM-памяти. И будет очень обидно, если из-за отладочных функций и макросов не будет умещаться основная программа.

Резерв скорости.

Естественно, подпрограммы вывода отладочной информации не выполняются мгновенно. Может получиться так, что из-за ее вывода программа будет просто не успевать отрабатывать свой алгоритм. Это нужно учитывать как при выборе тактовой частоты контроллера, так и при написании отладочных подпрограмм.

4.3. Функции-заглушки и функции-тестеры.

Заглушки.

В процессе тестирования могут пригодиться так называемые заглушки – пустотелые функции, соответствующие спецификации, но подменяющие вычисления (или результаты какой-то другой операции) заведомо правильным результатом. Когда это может оказаться полезным:

- Когда при отладке нужны результаты работы еще ненаписанных подпрограмм.
- При тестировании в симуляторе, когда нет возможности работать с какими-то внешними устройствами (например, GPS);
- При тестировании кода, содержащего долго выполняющиеся функции, не участвующие в тесте подпрограммы (типичный пример – задержки)

Тестеры.

Когда мы пишем новую функцию, мы хотим убедиться в ее работоспособности. Есть смысл писать небольшие функции, которые будут проводить тестирование новой, многократно запуская ее, передавая ей тестовые параметры. Дальнейший результат можно, в зависимости от назначения функции, наблюдать визуально или сравнивать с шаблоном. Такие функции могут формировать для нашей программы последовательность входных параметров, которая может возникнуть в реальной обстановке. Это также может оказаться особенно полезным, когда программа отлаживается или тестируется в симуляторе. Важно, чтобы при тестировании функция запускалась именно на рабочем контроллере или его аналоге (а не на Builder C++, Visual C++ и пр., на которых можно отлаживать только макет функции), т.к. каждый компилятор имеет свои особенности (типы данных, автоматическое преобразование типов, работа со стеком и пр.); каждый процессор – свои: архитектура (Гарвадрская и Неймоновская), объем доступной памяти и т.д.

Самое сложное место в этой схеме – шаблон. Ведь мы писали новую функцию именно для того, чтобы она нам этот результат вычисляла. Где брать шаблон для сравнения? Есть три пути:

1. В некоторых случаях есть возможность результат для сравнения формировать сторонней функцией. Например, известно, что функция `printf` из-за своей универсальности довольно громоздкая, и далеко не в каждом случае программист может позволить себе ее использовать, отняв у контроллера чуть ли не половину памяти и несколько тысяч тактов. Поэтому частое явление, когда программист пишет свою mini-`printf`, удовлетворяющую требованиям спецификации конкретной программы. Для проверки работоспособности своей функции он на этапе тестирования может пользоваться результатами работы встроенной функции `printf`. Причем эта функция может тестироваться в

автоматическом режиме. Такой же способ подойдет для проверки математических функций.

2. Входные параметры и результат вбивать вручную. Довольно трудоемкий процесс, поэтому в первую очередь желательно тестировать граничные значения.
3. Третий вариант – брать данные из вне (например, в автоматическом режиме с PC)

4.4. Компиляция

Серьезная ошибка, которую допускают программисты – это игнорирование предупреждений компилятора. Более того, к ним относятся как к назойливым мухам: «Ну, наконец-то собрал программу. Только своими дурацкими предупреждениями забил всю статистику, что ничего не прочтешь!» А между тем, компилятор ругается не просто так. Тут он нам не враг, а помощник. Он сообщает, что в программе есть двояко толкуемые конструкции, которые он транслировал на свое усмотрение. Дело в том, что ошибку можно совершить, даже написав все правильно с точки зрения языка. Например, в программе могут встретить выражения вида:

```
if (a == b) ...;
if (a = b)  ...; // Warning: Assignment inside relational expression
```

В зависимости от того, что мы хотим сделать, мы можем применить первое или второе выражение в скобках оператора `if`. В первом выражении мы проверяем переменные **a** и **b** на равенство, в то время как во втором мы проверяем переменную **a** на 0 после присваивания ей значения из переменной **b**. В общем случае второй записью пользоваться не рекомендуется, т.к. гораздо нагляднее:

```
a = b;
if (a) ...;
```

Потому-то компилятор и выдает нам предупреждение, что в выражении отношения имеется оператор присваивания, давая тем самым понять, что, возможно, мы имели в виду оператор отношения “==”. (Иногда, все же, есть смысл использовать вторую запись, когда, например, код критичен ко времени выполнения.)

Также частым следствием предупреждения компилятора является неаккуратное отношение к приведению типов. Например, в том же операторе сравнения:

```
signed   int  a;
unsigned int  b;
…
if (a > b) …; // Warning: signed and unsigned comparison
```

мы получим предупреждение, если одна переменная знаковая, а вторая – нет. Это предупреждение говорит не о том, что «осторожно! Может случиться коллизия», а о том, что программистом неправильно выбраны типы. Если же программист уверен в правильности, то он должен сделать приведение типов вручную:

```
signed   int  a;
unsigned int  b;
…
if ((signed long) a > (signed long) b) …;
```

Тем самым давая компилятору понять, что программист в курсе неправильно выбранных типов. Обратим внимание на то, что приведение типов производится к знаковому типу большей разрядности.

(Вообще же, повторюсь, с выражениями, где участвуют переменные различных типов, нужно быть осторожнее. Например, MPLAB C18 не выдаст предупреждения на наш пример, что в свою очередь приведет к неправильному поведению программы, если **a** будет иметь отрицательное значение.)

Что делать, если компилятор выдал предупреждение?

В первую очередь нужно определиться, почему он выдал предупреждение. Если его текст не понятен, то нужно обратиться к документации на компилятор и прочитать больше информации (часто в описании предупреждений приводятся наиболее частые причины их возникновения). Во вторую – привести код к однозначно интерпретируемому виду.

4.5. Вывод отладочной информации

При формировании отладочной информации нужно помнить, что информация об ошибках, выводимая для пользователя, и информация о поведении программы, выводимая для программиста, - это абсолютно разные вещи. Причем как по содержанию, так и по способу информирования.

Не нужно пользователя снабжать шестнадцатеричными кодами на дисплее, а по телефону потом разъяснять, что это «переменная режима работы приняла странное значение, сейчас подумаю, почему такое произошло». Также не нужно в конечном варианте программы оставлять всяческие вспыхивания светодиодов и взвизгивания пьезодинамика, которые при отладке говорили программисту о каком-то поведении программы или о ходе вычислений (а потом по телефону объяснять, что если лампочка мигнула 3 раза по 50 мс, то это не сошлась контрольная сумма). Т.е. вся отладочная информация, которая будет выводиться для программиста, должна быть скрыта от пользователя (запись в EEPROM, передача кода ошибки на центральный пульт, если такой есть).

4.6. Блокировка вывода отладочной информации

Нужно помнить, что возможность вывода отладочной информации, скорее всего, придется оставить в контроллере на всю жизнь. Иногда и отлаженные и переотлаженные программы могут содержать ошибки. Тем не менее, нужно иметь возможность отключать:

- вывод отладочной информации (как всей скопом, так и отдельных узлов);
- заглушки
- тестеры (их полезно оставлять в тексте программы на случай проявлений ошибок в период эксплуатации)

Блокировать можно двумя способами:

- **на этапе компиляции**, блокируя все отладочные узлы условными директивами компиляции:

```
#define DEBUG_ENABLE
    ...
    /*D*/ #ifdef DEBUG_ENABLE
    /*D*/ PIN_DEBUG = 1;
    /*D*/ #endif
```

  (тогда, закомментировав определение DEBUG_ENABLE, мы удаляем из кода всю отладочную составляющую). Такой способ экономит ресурсы контроллера.

- **на программно-аппаратном уровне** (например, контролируя состояние порта):

```
   if (PIN_DEBUG_ENABLE) PIN_DEBUG = 1;
```

(тогда мы можем включать/отключать работу отладочных подпрограмм в ходе выполнения.) Такой способ требует ресурсы, но позволяет отлаживать устройство в реальных условиях.

То же самое касается блокировки функций-заглушек и функций-проверок. Например:

```
#define DUMMY_GPS_IN
...
char * GetGPSData (void)
{
#ifdef DUMMY_GPS_IN
    char test_string[] = "$GPRMC,A,123456,…";
    return test_string;
#else
    /* Здесь код для работы с реальными данными от GPS */
#endif
}
```

В последнем примере иногда можно обойтись и без `#else`, т.к. внутри этого блока окажется довольно большой код, что неудобно.

Разумеется, такие блоки в коде следует визуально выделять.

4.7. Резервные копии

По ходу сопровождения (а иногда и разработки) программы нужно сохранять резервные копии, отмечая даты, номера версий, сделанные изменения и версию компилятора. Причем версию следует сохранять целиком, вместе с проектными файлами и HEX'ом. Бывает так, что незначительное изменение приводит к потере работоспособности кода, и, не имея резервных копий, зачастую трудно установить какая именно из последних модификаций привела к таким последствиям.

Однако сохранять целиком всю версию при каждой небольшой модификации довольно накладно (и по времени, и по трудоемкости). Поэтому для решения подобных задач удобно пользоваться специальными инструментами: системами контроля версий (VCS - version control systems).

Системы контроля версий обеспечивают два основных процесса:

- взаимодействие группы разработчиков, работающих над одним проектом
- сохранение текущих "срезов" проекта в базе данных.

Существуют различные реализации VCS: коммерческие и бесплатные, использующие центральный сервер и с распределенным хранением. Одна из самых распространенных - бесплатная система управления версий **Subversion** (http://ru.wikipedia.org/wiki/Subversion) с открытым исходным кодом. Именно ее можно порекомендовать для начального ознакомления с VCS.

Subversion (или SVN) является клиент-серверной системой. База данных проекта или репозиторий хранится на сервере, а разработчик работает с локальной копией проекта, которую "отдает" ему приложение-клиент. Поработав с локальной копией, разработчик сохраняет на сервере изменения проекта - таким образом обеспечивается безопасная работа в группе и полный контроль над ходом работы. В любой момент можно извлечь один из "срезов" (в терминологии SVN - ревизий) проекта и посмотреть всю историю изменений, а также произвести сравнение различных версий.

Наиболее популярная программа клиент Subversion для Windows - **TortoiseSVN** (`http://tortoisesvn.tigris.org`).

5. Список литературы

1. Э. Йодан «Структурное проектирование и конструирование программ», М. Мир, 1979
2. Б. Керниган, Р. Пайк «Практика программирования»
3. А. Голуб «Веревка достаточной длины, чтобы… выстрелить себе в ногу»
4. http://micrium.com/download/an2000.pdf - правила форматирования текста программ от micrium (на русском http://andromega.narod.ru/doc/micrium_an_2000_rus.pdf)
5. С. Макконнелл «Совершенный код», Русская редакция, 2005
6. http://www.pic24.ru/doku.php/articles/mchp/c30_atomic_access - статья об атомарном доступе
7. http://www.devdoc.ru/index.php/content/view/debugging_p1.htm - статья, посвященная отладке и тестированию приложений C++. Хоть и не имеет отношения к контроллерам, содержит много полезных советов.

Виктор Тимофеев, ноябрь, 2009

osa@pic24.ru