Багам вход воспрещен

Стандарт кодирования на языке C для встраиваемых систем

Майкл Барр

BARR group
The Embedded Systems Experts®

Издание: BARR-C: 2018 | barrgroup.com

Оглавление

Майкл Барр

Стандарт кодирования на языке C для встраиваемых систем

Авторские права: © 2018 Integrated Embedded, LLC (dba Barr Group). Все права защищены

Издано: Barr Group 20251 Century Blvd, Suite 330 Germantown, MD 20874

Эта книга может быть приобретена в печатном и электронном виде. Также доступна свободно распространяемая онлайн-версия. Больше информации доступно по ссылке: https://barrgroup.com/coding-standard.

Несмотря на все меры, предпринятые при подготовке данного документа, издатель и автор не несут никакой ответственности за ошибки и неполноту информации, а также за ущерб, вызванный её использованием. Соблюдение правил кодирования из данного документа не гарантирует отсутствие ошибок в разрабатываемом ПО и не накладывает на издателя и автора каких-либо юридических обязательств. Безопасность и надежность вашего ПО является вашей ответственностью.

«Barr Group» (включая логотип компании), «The Embedded Systems Experts» и «BARR-C» являются товарными знаками или зарегистрированными товарными знаками Integrated Embedded, LLC. Все остальные товарные знаки, используемые в данном документе, являются собственностью их законных владельцев.

Лицензия документа

Получая защищенный авторским правом «Стандарт кодирования на языке С для встраиваемых систем» от Barr Group («Документ»), вы принимаете условия настоящей Лицензии на документ («Соглашение»).

1. ПРЕДОСТАВЛЯЕМЫЕ ПРАВА. С учетом предоставленных каждой стороной другой стороне юридически действительных ценных встречных удовлетворений, получение и достаточность которых подтверждается настоящим Договором, Barr Group предоставляет вам следующую лицензию на использование данного Документа: вы можете опубликовать его исключительно для внутреннего использования и для применения вашими сотрудниками в рамках решения их рабочих задач. Получая копию данного Документа, вы в явной форме принимаете настоящее Соглашение.

2. ЗАЩИЩЕННЫЕ ПРАВА. За исключением случаев, явно разрешенных в настоящем Документе, вам не предоставляется никаких прав на Документ. Barr Group (от своего имени и имени своих аффилированных лиц) сохраняет за собой все остальные права на Документ в любой точке мира. В частности, вы признаете и соглашаетесь с тем, что: (i) Barr Group владеет авторскими правами на Документ; (ii) Документ и любые его производные не могут быть опубликованы или переданы другим лицам, за исключением случаев, разрешенных в п. 1 выше; (iii) вы должны принять соответствующие меры, касающиеся ограничений на публикацию и совместное использование, в том числе – включение соответствующих знаков авторского права и примечания об ограничениях публикации и передачи Документа, как описано в настоящем Соглашении. Если вы не включаете эти обязательные маркировки и обозначения авторских прав, то вы соглашаетесь с тем, что такие копии являются несанкционированными копиями материалов, защищенных авторским правом, и что эти несанкционированные копии нарушают авторские права Barr Group на Документ. Если вы готовите используете в своем документе, производной работе от данного Документа или его переводе фрагменты из данного Документа, то вы соглашаетесь с тем, что ваши права на эти новые документы, производные документы или переводы ограничены вашим личным (например, в рамках вашей компании) использованием.

3. ОТКАЗ ОТ ОБЯЗАТЕЛЬСТВ И ОГРАНИЧЕНИЕ ОТВЕТСТВЕННОСТИ. Вы соглашаетесь не допускать причинение ущерба, защищать и возмещать ущерб в случае его причинения Barr Group, ее владельцам и должностным лицам, сотрудникам и субподрядчикам, в полной мере, разрешенной законом, по любым искам, предъявленным в отношении данного Документа или его использования. Вы несете полную ответственность за определение того, является ли содержание Документа и его производных безопасным и подходит для ваших задач. Кроме того: (a) ДОКУМЕНТ ПРЕДОСТАВЛЯЕТСЯ «КАК ЕСТЬ» БЕЗ ЗАЯВЛЕНИЙ ИЛИ ГАРАНТИЙ ЛЮБОГО РОДА. В МАКСИМАЛЬНОЙ СТЕПЕНИ, РАЗРЕШЕННОЙ ПРИМЕНЯЕМЫМ ЗАКОНОДАТЕЛЬСТВОМ, BARR GROUP ПРЯМО ОТКАЗЫВАЕТСЯ ОТ ЛЮБЫХ ГАРАНТИЙ, ПРЕДУСМОТРЕННЫХ ЗАКОНОМ, ЯВНЫХ ИЛИ ПОДРАЗУМЕВАЕМЫХ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ, ЛЮБЫЕ ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ КОММЕРЧЕСКОЙ ПРИГОДНОСТИ, ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЗАДАЧИ И НЕНАРУШЕНИЯ ПРАВ В ОТНОШЕНИИ ДОКУМЕНТА. (b) Ни при каких обстоятельствах Barr Group не несет ответственность за любой ущерб (включая, но не ограничиваясь, особый, случайный, косвенный или непрямой ущерб здоровью, любые претензии третьих лиц, упущенную выгод, прерывание бизнес-процессов, потерю коммерческой информация или любые другие материальные убытки, гонорары адвокатов и судебные издержки), возникший в результате или в связи с использованием данного Соглашения

или Документа (независимо от того, знала ли или нет компания Barr Group о возможности любого подобного ущерба) или заявлением любой третьей стороны о том, что Документ, предоставленный вам по настоящему Соглашению, нарушает какие-либо авторские права или товарные знаки и права или неправомерно использует любую коммерческую тайну любой третьей стороны.

4. УПРАВЛЯЮЩЕЕ ПРАВО, ЮРИСДИКЦИЯ И ПОДСУДНОСТЬ. Настоящее Соглашение должно толковаться в соответствии с законодательством штата Мэриленд безотносительно противоречий в юридических положениях. Любой иск, вытекающий из настоящего Соглашения (или связанный с ним и относящийся к нему) должен быть передан либо в государственный суд общей юрисдикции округа Монтгомери, либо в Окружной суд США округа Мэриленд, который, по соглашению сторон, является местом и исключительной площадкой для рассмотрения любого дела или разногласия, вытекающие из настоящего Соглашения или связанного с ним. Каждая сторона безоговорочно подчиняется юрисдикции этих судов и отказывается от любых возражений против юрисдикции или места проведения судебных заседаний. В случае победы Barr Group в каком-либо иске, связанным с настоящим Соглашением, вы должны полностью оплатить Barr Group все судебные издержки, включая, помимо прочего, разумные гонорары для адвокатов.

5. НЕЗАВИСИМОСТЬ ПОЛОЖЕНИЙ СОГЛАШЕНИЯ И ВСТРЕЧНОЕ УДОВЛЕТВОРЕНИЕ. Каждый пункт настоящего Соглашения считается отдельной. Если по какой-либо причине какой-либо пункт настоящего Соглашения будет признан недействительным, то это не должно наносить ущерб или иным образом влиять на возможность принудительного исполнения других частей настоящего Соглашения. Получая Документ, вы признаете и соглашаетесь с тем, что соображения, лежащие в основе данного Соглашения — это взаимные обещания и обязанности, касающиеся содержимого данного Документа, а также использования вами Документа и его распространения в соответствии с условиями и положениями настоящего Соглашения.

Вступление

Цель данного стандарта

Стандарт кодирования на языке С для встраиваемых систем от компании Barr Group специально разработан для уменьшения количества ошибок во встраиваемом программном обеспечении. Следуя правилам этого стандарта, инженеры-программисты не только могут снизить уровень опасности своих приборов для пользователей и уменьшить время на отладку ошибок, но и сделать свой код более переносимым и простым в поддержке. В сумме эти факторы позволяют значительно снизить стоимость разработки высоконадежных встраиваемых приложений.

Стандарт кодирования «*BARR-C*» отличается от других стандартов кодирования. Приведенные в нем правила отбирались не по стилистическим предпочтениям автора, а по их возможности свести к минимуму ошибки ПО. Если для какого-то аспекта существовало несколько различных подходов – то из них выбирался тот, который способен предотвратить наибольшее число ошибок. Например, правила расстановки фигурных скобок были выбраны таким образом, чтобы уменьшить как можно больше ошибок во всей программе.

Отдельные правила, от следования которым объективно ожидается снижение числа ошибок, помечены иконкой «Багам вход воспрещен»:

Вполне очевидно, что ни один стандарт кодирования не позволяет полностью устранить вероятность появления ошибок в процессе разработки. Взаимодействие между аппаратным и программным обеспечением, а также между различными системами, сложны по своей природе. Даже если разработкой занимается команда опытных программистов с превосходными навыками кодирования и следующая всем правилам данного стандарта, то ошибки всё равно могут возникать из-за:

- ошибок в требованиях;
- неправильного понимания требований;
- недоработок в архитектуре системы и/или ПО;
- недостаточной обработки аппаратных сбоев или других исключительных ситуаций в процессе работы системы;
- и т.д.

Еще одним важным стимулом использования данного стандарта кодирования является увеличение читабельности и переносимости кода. Это позволяет упростить сопровождение кода и обеспечить возможность его повторного использования. Использование полного набора правил стандарта (то есть не только тех, которые нацелены непосредственно на устранение возможных

ошибок) принесет пользу команде разработчиков и компании в целом за счет уменьшения времени, требуемого на понимание кода, написанного коллегами и бывшими сотрудниками.

Мы рекомендуем применять стандарт кодирования BARR-C в рамках более широкого комплекса мер по улучшению процесса разработки встраиваемого ПО и обеспечения его качества в вашей организации. Безусловно, это не выглядит таким важным, как обеспечение безопасности пользователей ваших продуктов, но следует осознавать значимость системной и программной архитектуры в контексте уменьшения числа причин потенциальных сбоев в процессе работы оборудования, а также профессиональной подготовки программистов[1]. Как минимум, ваш процесс разработки должен включать не только использование стандарта кодирования, но и применение инструментов контроля версий и отслеживания дефектов (например, Jira), формализованных обзоров архитектуры и дизайна ПО, code review, а также автоматическое сканирование исходного кода с помощью одного или нескольких инструментов статического анализа.

Основные принципы

Чтобы акцентировать внимание на действительно важных вопросах и не зацикливаться на тех моментах, которые часто воспринимаются разработчиками как личные стилистические предпочтения, данный стандарт кодирования был разработан в соответствии со следующими принципами:

1. Код не является собственностью разработчика. Разработка ПО оплачивается работодателем или клиентом, поэтому конечный продукт должен соответствовать сформулированным ими требованиям.

2. Дешевле и проще предотвратить появление в коде ошибки, чем потом тратить время и ресурсы на ее поиск и устранение. Основной стратегией в данной ситуации является написание такого кода, для которого компилятор, компоновщик и статический анализатор могут обнаружить ошибку автоматически – еще до запуска программы.

3. Сложно сказать, хорошо ли это или плохо (ну, честно говоря, плохо) – но стандарт ISO C допускает значительные отличия между различными реализациями компиляторов[2]. В стандарте можно встретить достаточно много моментов, связанных с «поведением, определяемым реализацией» (implementation-defined behavior), «неуточняемым поведением» (unspecified behavior), «неопределенным поведением» (undefined behaviour), а также с регионально-зависимыми настройками. В результате приложение, скомпилированное из одного и того же исходного кода с помощью разных компиляторов (соответствующих стандарту ISO C), может работать по-разному. Эта «серая зона» стандарта языка C значительно снижает переносимость исходного кода, если на этапе разработки вопросам портируемости не уделялось особого внимания.

4. Надежность, читабельность, эффективность и, в некоторых случаях, переносимость кода важнее предпочтений программиста.

[1] Если из-за неисправности/ошибок в работе вашего оборудования или несоблюдения правил безопасности пользователи могут получить травмы или погибнуть – следует соблюдать соответствующие стандарты по разработке функционального безопасного (safety) ПО. Данный документ НЕ ЯВЛЯЕТСЯ таким стандартом.
[2] См., например, [C90] и [C99]

5. Существует много источников ошибок в программном обеспечении. Часть ошибок будет допущена командой разработки. Программисты, которые будут поддерживать, расширять или переиспользовать этот код, могут добавить дополнительные ошибки – в частности, из-за неправильного понимания исходного кода.

- число и уровень критичности ошибок, вносимых первоначальной командой разработчиков, могут быть уменьшены за счет строго соблюдения определенных правил кодирования – например, размещения констант слева от оператора сравнения (==);
- число и уровень критичности ошибок, вносимых на этапе сопровождения, может быть снижено на этапе разработки – например, путем выбора типов данных, которые на любых платформах будут иметь одинаковый размер (например, int32_t) и позволят избежать неожиданного переполнения;
- число и уровень критичности ошибок, вносимых на этапе сопровождения, может быть снижено за счет строгого использования осмысленных комментариев и определенных стилистических приемов – что позволит новым разработчикам правильно понять, как использовать переменные, функции и модули исходной программы.

6. Чтобы эффективно выполнять свою задачу – стандарт кодирования должен быть применимым в реальной жизни. Если два или более взаимоисключающих правила в равной степени предотвращают появление ошибок, то должно использоваться то правило, которому проще следовать.

При отсутствии в данном документе какого-либо правила по интересующему вас вопросу или противоречии приведенных правил с принятым в вашей команде стандарте кодирования – принимать решение следует, основываясь на описанных выше принципах.

MISRA C

Стандарт *MISRA C:2012 – Guidelines for the Use of the C Language in Critical Systems* (см. [MISRA-C]) определяет подмножество языка C, которое является более безопасным и в то же время более строгим по сравнению с правилами стандарта BARR-C.

Если вы разрабатываете оборудование, из-за ошибок в котором люди могут получить травмы или погибнуть, то важно изучить рекомендации MISRA C и добавить их в стандарт кодирования вашей команды разработки. Стандарт MISRA C имеет 20-летнюю историю; актуальная редакция стандарта – 3-я. Скорее всего, авторы MISRA C лучше вас осведомлены о рисках использования языка C в системах, где вопросы функциональной безопасности являются критически важными.

При разработке данного документа были предприняты все усилия для того, чтобы правила BARR-C можно было сочетать с правилами MISRA-C:2012. В частности:

- правила данного документа, которые касаются ограничений на использование отдельных элементов стандарта языка C (например, ключевых слов *goto* и *register*), никогда не являются более строгими, чем правила MISRA C. Иными словами, MISRA C представляет собой подмножество BARR-C, который сам является подмножеством стандарта языка C;
- правила данного документа, которые касаются стилистических ограничений (например, ограничений на имена переменных и функций), никогда не противоречат правилам MISRA C. Другими словами, BARR-C включает в себя стилистические рекомендации, дополняющие стандарт MISRA C, который не содержит никаких рекомендаций, связанных исключительно со стилем кодирования.

Опрос 2018 года показал, что более 40% программистов встраиваемого ПО совместно используют стандарты MISRA-C и BARR-C как основу своих внутренних корпоративных стандартов разработки.

C++ vs C

Хотя в названии этой книги упоминается только язык C, разработчики встраиваемого ПО, которые используют язык C++ (или смесь обоих языков), также могут сократить число ошибок в своих программах, следуя приведенным рекомендациям. Это связано с тем, что синтаксис C++ основан на синтаксисе C и множество строк C++ кода фактически являются кодом на языке C.

Однако важно отметить, что C++ является значительно более сложным и обширным по возможностям языком, чем C, и содержит ряд функций, не имеющих эквивалента в C. Если вы планируете применять правила BARR-C при программировании на C++, то вам следует серьезно подумать о переходе на другой стандарт кодирования – например, [MISRA-C++], [Sutter] и/или [Holub].

В рамках данной книги основной акцент сделан на языке C, так как он является основным языком для приблизительно 70% профессиональных разработчиков встраиваемого ПО. Изучение отраслевых опросов, проведенных между 2005 и 2018 годом, показывает, что C не только был наиболее широко используемым языком для разработки встраиваемого ПО, но и увеличил свою

долю на рынке с ≈50% до ≈70% за этот период. В сообществе разработчиков встраиваемого ПО есть мнение, что пик популярности C++ пришелся на 2006 год.

Руководство по использованию стандарта

Соблюдение всех правил данного стандарта считается обязательным. Крайне желательно, чтобы была возможность обнаруживать код, который не соответствует стандарту – в первую очередь, автоматически (с помощью статических анализаторов), во-вторых – в процессе code review. Если такой возможности нет, то несоответствия могут быть обнаружены неформализованным образом (например, при изучении одним разработчиком кода другого разработчика). После обнаружения код, который не соответствует правилам стандарта, должен быть исправлен.

На рынке ПО доступны коммерческие инструменты статического анализа, которые можно использовать для автоматической проверки соответствия кода правилам данного и других стандартов кодирования. Используемый стандарт кодирования должен быть указан в настройках статического анализатора. Стандарт кодирования данного документа предлагается называть «BARR-C:2018».

В случае перехода с одного стандарта кодирования на другой необходимо принять решение, что делать с унаследованной кодовой базой. Немногие команды разработчиков имеют возможность провести рефакторинг всех ранее созданных ими библиотек.

Для работы с legacy-кодом мы можем озвучить следующие рекомендации:

- в большинстве случаев лучше оставить работающий legacy-код в покое («работает – не трогай»), если только речь не идет о жизни и здоровье ваших клиентов;
- если принято решение привести legacy-код в соответствие новому стандарту кодирования – то для каждого выбранного модуля или библиотеки изменения синхронно должны вноситься в заголовочный файл (.h) и в файл исходного кода (.c). Обычно стилистические изменения лучше вносить тогда, когда возникла необходимость в функциональных изменениях.

Обратите внимание, что изменения, связанные с изменением размеров отступов (пробелов и табуляции) должны быть сделаны не только в исходном коде, но и в инструменте контроля версий, чтобы обеспечить максимальную эффективность утилит сравнения файлов (diff и т.д.), созданных до и после этих изменений.

Отступления от стандарта

Весь код, отправляемый в релиз, должен соответствовать всем приведенным в документе правилам, за исключением случаев, когда конкретные отклонения были обсуждены, утверждены и задокументированы.

Численные ограничения, упомянутые в ряде правил (например, количество пробелов в отступе или максимальное количество строк кода в функции) можно изменить на другие, более подходящие для конкретных инструментов разработки. Обычно конкретное значение не играет роли в подобных правилах.

При работе с файлами исходного кода любые отклонения от правил допустимы только в случае их согласования с руководителем проекта. Имя руководителя, причины отклонения от стандарта и особенности кода, подверженного отклонениям, должны быть тщательно задокументированы.

Единичное отклонение в коде функции может быть задокументировано в виде комментария в заголовке или реализации данной функции. Отклонение на уровне модуля лучше задокументировать в первых строках файла исходного кода данного модуля.

Адаптация стандарта

Авторские права на этот документ, включая набор и порядок описанных в нем правил, принадлежат Barr Group. Мы разрешаем отдельным командам разработчиков, целым компаниям и другим лицам использовать часть или все правила данного документа в рамках своего стандарта кодирования. Безусловно, мы рады, что некоторые читатели предыдущих изданий уже так и поступили и надеемся, что в будущем число таких разработчиков увеличится. При ссылке на этот документ как на источник информации можно использовать название «Стандарт кодирования на языке C для встраиваемых систем от Barr Group» или же просто «BARR-C:2018».

Чтобы помочь группам разработчиков адаптировать этот стандарт кодирования в соответствии с потребностями конкретной компании, редактируемая версия данного документа доступна для лицензирования и загрузки по ссылке https://barrgroup.com/embedded-systems/books/embedded-c-coding-standard.

Ваши юридические обязательства в отношении использования этого документа, защищенного авторским правом, приведены в начале этой книги.

Благодарности

Несмотря на то, что автором данного документа указан я (Майкл Барр), разработка и поддержка стандарта кодирования на языке C для встраиваемых систем является плодом совместной работы различных специалистов, начавшейся более 10 лет назад, и в которой приняли участие большинство нынешних сотрудников Barr Group, а также множество других участников сообщества разработчиков встраиваемого ПО. Я особенно признателен Соломону Сингеру и Джо Перрету за их помощь в работе над первым изданием (2008) этой книги, которая позволила мне воплотить ее в жизнь, Гэри Стрингхэму за тесное сотрудничество в работе над изданием 2018 года и всем тем, кто присылал свои комментарии и предложения по доработке правил, вычитывал черновики любого из изданий и предоставлял обратную связь на протяжении всех этих лет.

1. Основные правила

1.1. Какую версию стандарта языка C использовать?

Правила:

a. Весь программный код должен соответствовать версии C99[3] стандарта ISO C.

b. В случае использования компилятора C++ следует установить нужные опции, чтобы ограничить возможности языка выбранной версией (C99);

c. Использование дополнительных ключевых слов, не определенных в стандарте языка, директивы *#pragma* и ассемблерных вставок должно быть сведено к необходимому для конкретной задачи минимуму и ограничено небольшим числом драйверов, напрямую взаимодействующих с аппаратным обеспечением;

d. Директива *#define* не должна использоваться для переопределения ключевых слов языка.

Пример:

```
#define begin {   // Не делайте что-то вроде этого
#define end }     // ...и этого
...
for (int row = 0; row < MAX_ROWS; row++)
begin
...
end // позвольте C быть C, а не делайте из него тот язык, который вы предпочитаете
```

Объяснение: чтобы четко определить остальные правила, необходимо выбрать конкретную версию стандарта языка программирования.

Применение: проверка выполнения этого правила должна производиться компилятором (на основании заданных опций компиляции) и во время code review.

[3] Компиляторы, совместимые с C99, имеют множество серьезных улучшений по сравнению со старыми версиями – например, однострочные комментарии в стиле C++ (//), булевский тип данных, целочисленные типы фиксированного размера (с постфиксом _t – int16_t и т.д.), встраиваемые (inline) функции, возможность объявлять локальные переменные в любом фрагменте файла исходного кода и т.д.

1.2. Ограничение на длину строк кода

Правила:

а. Максимальная длина одной строки кода должна быть ограничена 80 символами.

Объяснение: время от времени в процессе code review и других процедур приходится работать с распечатками программ. Чтобы этот процесс был эффективным – такие распечатки не должны содержать переносов в произвольных местах или потерянного (обрезанного по правой границе листа) текста. Ограничение на длину строк кода также упрощает построчное сравнение нескольких версий одного и того же кода, одновременно отображаемых на экране.

Применение: проверка выполнения этого правила должна автоматически производиться на этапе сборки.

1.3. Фигурные скобки

Правила:

a. Блоки кода (также называемые «составными операторами»), размещаемые после операторов *if*, *else*, *switch*, *while*, *do* и *for*, должны быть заключены в фигурные скобки. Одиночные операторы и пустые операторы, размещенные после этих ключевых слов, также должны заключаться в фигурные скобки.

b. Открывающая фигурная скобка ({) должна быть размещена на отдельной строке перед первой строкой блока кода. Закрывающая фигурная скобка (}) должна быть размещена на отдельной строке после последней строки блока кода и на том же уровне (с тем же отступом), что и открывающая скобка.

Пример:

```
{
   if (depth_in_ft > 10) dive_stage = DIVE_DEEP; // Это корректно...
   else if (depth_in_ft > 0)
       dive_stage = DIVE_SHALLOW;               // ...и это тоже.
   else
   {                                 // Но использовать скобки всегда безопаснее.
       dive_stage = DIVE_SURFACE;
   }
   ...
}
```

Объяснение: использование отдельных операторов или пустых операторов, не заключенных в фигурные скобки, связано со значительным риском. Такие фрагменты кода часто становятся ошибочными, когда другой размещенный рядом с ними код меняется или оказывается закомментирован. Этот риск полностью устраняется использованием фигурных скобок. Размещение фигурных скобок на отдельных строках и с одним уровнем отступа позволяет легко визуально определить потерю одной из скобок.

Применение: проверка наличия открывающей фигурной скобки после оператора *if*, *else*, *switch*, *while*, *do* и *for* и идентичного отступа для парных скобок должны автоматически выполняться во время сборки.

1.4. Круглые скобки

Правила:

a. Не полагайтесь исключительно на приоритеты выполнения операторов, определенные в стандарте языка С, так как они могут быть неочевидны для тех, кто будет поддерживать ваш код. Чтобы внести ясность в порядок выполнения операторов – используйте круглые скобки (и/или разбивайте длинные выражения на сколько строк кода).

b. Если операнды операторов логического И (&&) и ИЛИ (||) не являются одиночными переменными или константами – то они должны быть заключены в круглые скобки.

Пример:

```
if ((depth_in_cm > 0) && (depth_in_cm < MAX_DEPTH))
{
    depth_in_ft = convert_depth_to_ft(depth_in_cm);
}
```

Объяснение: стандарт языка С включает в себя множество операторов. Правила, определяющие приоритет выполнения операторов, довольно сложны (число уровней приоритета превышает дюжину) и не всегда очевидны для конкретного разработчика. Если вы сомневаетесь, в каком порядке будут выполнены ваши операторы, то лучше указать компилятору этот порядок в явном виде с помощью круглых скобок.

Применение: проверка выполнения этих правил должна производиться во время code review.

1.5. Распространенные аббревиатуры

Правила:

a. Аббревиатур и акронимов следует избегать – за исключением тех, которые являются общепринятыми в инженерном сообществе.

b. Список аббревиатур и акронимов, используемых в конкретном проекте, должен быть задокументирован и регулярно обновляться в процессе разработки.

Пример: в приложении А приведен пример списка аббревиатур и их расшифровок.

Объяснение: программисты слишком охотно используют загадочные аббревиатуры и акронимы в своем коде (и резюме!). Тот факт, что вы в данный момент помните, что означает «ZYZGXL», не значит, что это поймут разработчики, которые будут читать или поддерживать ваш код в будущем.

Применение: проверка выполнения этих правил должна производиться во время code review.

1.6. Преобразования типов

Правила:

а. Каждое преобразование типов должно сопровождаться комментарием, поясняющим, как обеспечивается корректность преобразования на всем диапазоне значений исходного типа.

Пример:

```
int
abs (int arg)
{
    return ((arg < 0) ? -arg : arg);
}
...
uint16_t sample = adc_read(ADC_CHANNEL_1);
result = abs((int) sample); // ПРЕДУПРЕЖДЕНИЕ: предполагается 32-битный int
```

Объяснение: преобразование типов опасно. В приведенном выше примере значение беззнакового 16-битного «сэмпла» может быть больше, чем верхний предел 16-битного знакового целого. В этом случае полученное в результате конвертации значение будет некорректным из-за переполнения. Стандарт ISO C допускает интерпретацию типа int как 16-битного знакового целого.

Применение: проверка выполнения этих правил должна производиться во время code review.

1.7. Ключевые слова, которых следует избегать

Правила:

a. Запрещается использовать ключевое слово *auto*.

b. Запрещается использовать ключевое слово *register*.

c. Не рекомендуется использовать оператор *goto*. Если этот оператор всё же используется – то он должен использоваться только для перехода к метке, расположенной в этом же или соседнем блоке кода.

d. Не рекомендуется использовать оператор *continue*.

Объяснение: ключевое слово *auto* – рудимент первых версий языка C. Использование ключевого слова *register* подразумевает, что программист умнее компилятора. В настоящее время нет причин использовать эти ключевые слова.

Операторы *goto* и *continue* всё еще являются частью языка, но их использование очень часто приводит к появлению спагетти-кода. В частности, использование *goto* для совершения «прыжков» по программе противоречит концепции потока управления, на которой основано структурное программирование. Использование *goto* для обработки исключений приемлемо, если это упрощает код и делает его более читабельным.

Применение: проверка наличия в коде запрещенных ключевых слов должна автоматически производиться на этапе сборки. В случае использования ключевых слов *goto* и *continue* – во время code review должна быть выполнена оценка целесообразности этого шага в данном конкретном случае с изучением возможных альтернативных вариантов.

1.8. Часто используемые ключевые слова

Правила:

a. Ключевое слово *static* должно использоваться для объявления всех функций и переменных, которые должны быть ограничены областью видимости модуля, в котором объявлены.

b. Ключевое слово *const* должно использоваться везде, где это возможно. Например:

i. При объявлении переменных, значения которых не должны изменяться после инициализации.

ii. При указании параметров функции, которые получают константные значения по ссылке (например, char const * param);

iii. При объявлении полей структур и объединений, значения которых не должны изменяться (например, при наложении структуры на регистры периферии);

iv. В качестве строго типизированной альтернативы *#define* при объявлении числовых констант.

c. Ключевое слово *volatile* должно использоваться везде, где это возможно. Например:

i. При объявлении глобальных переменных, которые должны быть доступны любой подпрограмме, обслуживающей прерывание;

ii. При объявлении глобальных переменных, которые должны быть доступны в двух или более потоках;

iii. При объявлении указателя на набор регистров периферии ввода-вывода (например, timer_t volatile * const p_timer);

iv. При объявлении счетчиков циклов задержки.

```
typedef struct
{
    uint16_t count;
    uint16_t max_count;
    uint16_t const _unused; // read-only register
    uint16_t control;
} timer_reg_t;
timer_reg_t volatile * const p_timer = (timer_reg_t *) HW_TIMER_ADDR;
```

Объяснение: ключевое слово *static* языка C имеет несколько значений. На уровне модуля глобальные переменные и функции, объявленные как статические, запрещены для доступа из других модулей. Таким образом, использование ключевого слова *static* позволяет избежать лишних связей между модулями.

Ключевые слова *const* и *volatile* еще более важны. Преимущество использования *const* заключается в принудительной защите компилятором от непреднамеренной записи данных, которые должны быть доступны только для чтения. Правильное использование *volatile* устраняет целый ряд трудно обнаруживаемых ошибок, предотвращая оптимизацию компилятора, которая запрещала бы операции чтения или записи в переменные или регистры[4].

Применение: проверка выполнения этих правил должна производиться во время code review.

[4] Наш личный опыт свидетельствует о том, что программисты, не знакомые с ключевым словом *volatile*, считают, что оптимизации от компилятора скорее что-то сломают, чем принесут пользу – и поэтому отключают их. Мы считаем, что подавляющее большинство встраиваемых систем содержат ошибки, связанные с отсутствием ключевого слова *volatile*, которые проявляются не сразу, а ждут своего часа. Такие ошибки обычно проявляются как странные «глюки» в произвольный момент работы или после внесения изменений в «проверенную» кодовую базу.

2. Комментарии

2.1. Формат комментариев

Правила:

a. Однострочные комментарии в стиле C++ (// текст комментария) являются разумной и приемлемой альтернативой традиционным комментариям в стиле C (/* текст комментария */);

b. Комментарии не должны включать токены препроцессора /*, // и \.

c. Комментарии не должны содержать код (даже временное помещение кода в комментарий не допускается).

i. Чтобы временно исключить блок кода из компиляции используйте директиву препроцессора *#if 0 … #endif*;

ii. Код, который используется для повышения детализации отладочной информации, должен быть помещен в условие *#ifndef NDEBUG … #endif*.

Пример:

```
/* Этот код должен был войти в состав сборки…
...
safety_checker();
...
/* ... но символ окончания комментария был пропущен. */
```

Объяснение: вложенные комментарии могут ввести в заблуждение рецензентов кода относительно того, какие фрагменты кода войдут в состав сборки, а какие будут исключены из компиляции. Мы осознанно используем макрос препроцессора NDEBUG, поскольку он также отключает макрос *assert()*.

Применение: проверка формата комментариев должна выполняться статическим анализатором. Но только рецензент во время code review может отличить закомментированный код от комментариев, содержащих примеры использования кода.

2.2. Расположение и содержание комментариев

Правила:

a. Все комментарии должны быть ясными и последовательными, без орфографических, грамматических и пунктуационных ошибок;

b. Самые полезные комментарии обычно размещены перед блоком кода, выполняющим фрагмент крупного алгоритма. После каждого такого блока должна быть размещена пустая строка. Комментарии различных блоков должны иметь одинаковый уровень отступа;

c. Избегайте пояснений для очевидных фрагментов кода. Исходите из предположения, что тот, кто будет читать ваш код, знаком с программированием на языке C. Например, комментарий после строки кода имеет смысл оставлять только в том случае, если по названию переменных и функций может быть непонятно, что происходит в данной строке, но короткий комментарий может внести в это ясность. Избегайте бесполезных и избыточных комментариев, например

```
numero <<= 2; // Сдвинуть numero на 2 бита влево
```

d. Число блоков комментариев и их размеры должны быть пропорциональны сложности кода, который они описывают;

e. Если алгоритм или детали реализации описываются в каком-либо документе (например, в спецификации, патенте или учебнике), то комментарий должен включать ссылку на этот документ. Эта ссылка должна включать в себя информацию, достаточную для того, чтобы читатель мог найти этот документ;

f. Если для документирования кода требуется блок-схема или другая диаграмма, то файл ее рисунка должен быть размещен в системе контроля версий рядом с файлом исходного кода, а в комментарии должно быть указано имя рисунка или название файла;

g. Все предусловия должны быть описаны в комментариях[5];

h. Каждый модуль и функция должны быть прокомментированы в стиле, который поддерживает одна из систем автоматического создания документации (например, Doxygen).

[5] Еще лучше разработать тесты или утверждения (assertions) по методике контрактного программирования. См, например: https://barrgroup.com/embedded-systems/how-to/design-by-contract-for-embedded-software

i. Используйте в комментариях следующие маркеры, записанные заглавными буквами:

i. «WARNING:» – предупреждения о рисках, связанные с изменением данного фрагмента кода. Например, значение задержки было определено опытным путем и, возможно, потребуется изменить его при переносе кода на другую платформу или изменений опций оптимизации компилятора;

ii. «NOTE:» – объяснения особенностей реализации фрагмента кода, отвечающие на вопрос «почему?» (ответы на вопросы «каким образом?» обычно размещены в комментариях к конкретным строкам кода). Например, часть кода драйвера может не соответствовать спецификации из-за присутствующих в ней ошибок, еще не внесенных в errata;

iii. «TODO:» – этот маркер указывает на то, что фрагмент кода всё еще находится в разработке и поясняет, что осталось сделать. Если это необходимо – перед маркером могут быть указаны инициалы разработчика (например, «MJB TODO:»).

Пример:

```
// Шаг 1: задраить люки.
for (int hatch = 0; hatch < NUM_HATCHES; hatch++)
{
   if (hatch_is_open(hatches[hatch]))
    {
        hatch_close(hatches[hatch]);
    }
}

// Шаг 2: поднять бизань-мачту.
// TODO: разработать API драйвера бизань-мачты.
```

Объяснение: соблюдение этих правил позволяет получить хорошие комментарии – а они являются неотъемлемой частью хорошего кода. Рекомендуется сначала добавить комментарии, описывающие поведение системы, а потом написать код, реализующий это поведение.

К сожалению, по мере внесений исправлений исходный код и документация могут рассинхронизироваться. Лучший способ избежать этого — держать документацию как можно ближе к коду. Каждый раз, когда возникает вопрос о работе фрагмента кода, который раньше считался понятным, вы должны добавить к этому коду новый комментарий.

Doxygen — полезный инструмент для автоматического создания пользовательской документации, описывающей модули, функции и параметры API. Тем не менее, исходный код всё равно должен быть прокомментирован, чтобы снизить затраты на его поддержку.

Применение: качество комментариев должно оцениваться во время code review. Рецензенты должны удостовериться, что комментарии точно описывают код, а также являются ясными, краткими и осмысленными. Автоматически создаваемая документация должна генерироваться при каждой сборке ПО.

3. Пробелы и отступы

3.1. Пробелы

Правила:

a. После ключевых слов *if*, *while*, *for*, *switch* и *return* должен присутствовать одиночный пробел (если после этих слов на данной строке есть еще какие-то символы);

b. Операторы присваивания =, +=, -=, *=, /=, %=, &=, |=, ^=, ~= и != должны отделяться одиночными пробелами с обеих сторон;

c. Логические операторы +, -, *, /, %, <, <=, >, >=, ==,!=, <<, >>, &, |, ^, && и || должны отделяться одиночными пробелами с обеих сторон;

d. Унарные операторы +, -, ++, --, ! , и ~ не должны отделяться от операнда пробелом (пример: i++, а не i ++);

e. Операторы взятия адреса и разыменования & и * должны отделяться одиночными пробелами с обеих сторон при объявлении переменных, а в остальных случаях не должны отделяться от операнда пробелом;

f. Символы ? и : в тернарном операторе должны отделяться одиночными пробелами с обеих сторон;

g. Операторы обращения к элементу структуры (. и ->) должны быть записаны без пробелов с обеих сторон;

h. Скобки оператора индексного доступа ([и]) должны быть записаны без пробелов с обеих сторон, кроме тех случаев, которые оговорены другими правилами;

i. Выражения в круглых скобках должны быть записаны без пробелов с обеих сторон скобок;

j. Левая и правая скобки оператора вызова функции должны быть записаны без пробелов с обеих сторон скобок за исключением объявления функции – в нем между именем функции и левой скобкой должен быть одиночный пробел, чтобы можно было легко найти место ее объявления;

k. После запятой, разделяющей аргументы функции, должен быть установлен одиночный пробел, за исключением случая, когда запятая является последним символом строки;

l. После каждой точки с запятой, разделяющей элементы оператора *for*, должен быть установлен одиночный пробел;

m. Перед точкой с запятой, завершающей выражение, не должно быть установлено пробела.

Пример: см. приложение D.

Объяснение: расстановка пробелов является важным аспектом оформления кода. Грамотное использование пустого пространства на экране снижает нагрузку на глаза и повышает вероятность обнаружения ошибки разработчиком или рецензентом.

Применение: программисты должны соблюдать эти правила и применять инструменты автоматического форматирования кода (например, GNU Indent).

3.2. Выравнивание

Правила:

a. При объявлении переменных их имена должны быть выровнены;

b. При объявлении полей структур и объединений их имена должны быть выровнены;

c. Операторы присваивания, размещенные в рамках одного блока кода с набором операций присваивания на соседних строках, должны быть выровнены относительно друг друга;

d. Символ «#» в директивах препроцессора всегда должен быть размещен в начале строки и не иметь отступов. При этом текст директив может быть выровнен (например, это касается выражений в директивах *#if* и *#ifdef*).

Пример:

```
#ifdef USE_UNICODE_STRINGS
# define BUFFER_BYTES 128
#else
# define BUFFER_BYTES 64
#endif
...
typedef struct
{
    uint8_t buffer[BUFFER_BYTES];
    uint8_t checksum;
} string_t;
```

Объяснение: выравнивание позволяет визуально выразить связность расположенных рядом строк кода. Последовательность строк объявления переменных с одинаковым выравниванием легко воспринимается как единый фрагмент кода. Пустые строки и различные варианты выравнивания следует использовать по мере необходимости, чтобы визуально отделить несвязанные блоки кода, которые расположены рядом друг с другом.

Применение: проверка выполнения этих правил должна производиться во время code review.

3.3. Пустые строки

Правила:

a. Каждый уровень отступа должен быть равен 4 пробелам. Отступы отсчитываются от начала строки;

b. Метки *case* в операторе *switch* должны быть выровнены. Код, размещенный внутри метки, должен иметь один уровень отступа (в рамках этого кода при необходимости могут быть использованы дополнительные уровни отступа);

c. Если длина строки кода превышает ограничение, то вторая и следующие строки, на которые будет перенесен код, должны иметь отступы, оформленные наиболее удобным для читателя образом.

Пример:

```
sys_error_handler(int err)
{
    switch (err)
    {
        case ERR_THE_FIRST:
        ...
        break;

        default:
        ...
        break;
    }
    // Здесь пропущен один отступ для повышения читабельности
    if ((first_very_long_comparison_here
        && second_very_long_comparison_here)
       || third_very_long_comparison_here)
    {
    ...
    }
}
```

Объяснение: размер отступа в 4 пробела выбран по следующим соображениям: меньшее количество пробелов в отступе увеличивает риск некорректного понимания кода, а большее количество пробелов увеличивает вероятность лишних переносов строк.

Применение: разработчики должны использовать инструменты для автоматического форматирования кода, которые позволяют настраивать величину отступов и изменение размеров уже существующих отступов. Этот инструмент должен запускаться для всех создаваемых или редактируемых файлов кода перед каждой сборкой.

3.4. Табуляция

Правила:

a. Символ табуляции (символ с ASCII-кодом 0x09) не должен использоваться в файлах кода.

Пример:

```
// Если табуляция нужна внутри строковой переменной – используйте спецсимвол '\t'
#define COPYRIGHT "Copyright (c) 2018 Barr Group.\tAll rights reserved."
// Отступы в исходном коде должны быть созданы пробелами, а не табуляцией.
void
main (void)
{
 // Если игнорировать это правило
   // то вы столкнетесь
// со странными
 // и несогласованными
    // отступами в коде и комментариях, созданными в разных редакторах/IDE.
}
```

Объяснение: ширина табуляции зависит от настроек текстового редактора и предпочтений конкретного разработчика, что создает множество головной боли при чтении кода в другом редакторе (например, в процессе code review).

Применение: каждый программист должен настроить свои инструменты таким образом, чтобы нажатие на TAB приводило к вставке нужного числа пробелов. Наличие символа табуляции в новом или редактируемом файле кода должно детектироваться в процессе автоматической проверки каждой сборки ПО.

3.5. Непечатаемые символы

Правила:

a. По возможности все строки исходного кода должны заканчиваться непечатаемым символом «LF[6]» (ASCII—код: 0x0A), а не парой «CR[7]»/«LF» (0x0D 0x0A);

b. Кроме «LF» допускается использование непечатаемого символа «FF[8]». Использование в коде других непечатаемых символов запрещается.

Пример: невозможно продемонстрировать в тексте пример, связанный с непечатаемыми символами.

Объяснение: последовательность «CR»/«LF» с большей вероятностью вызовет проблемы в кросс-платформенной среде разработки, чем одиночный символ «LF». Одна из возможных проблем связана с многострочными макросами препроцессора на платформе Unix.

Применение: по возможности текстовые редакторы разработчиков должны быть настроены на использование символа «LF» для переноса строк. Кроме того, последовательность «LF»/ «CR» в новом или редактируемом файле кода должна заменяться на одиночный «LF» в процессе автоматической проверки каждой сборки ПО.

[6] Символ переноса строки (Line Feed)
[7] Символ возврата каретки (Carriage Return)
[8] Символ смены страницы (Form Feed)

4. Модули кода

4.1. Названия

Правила:

a. Названия модулей должны включать в себя только символы нижнего регистра, цифры и символ подчеркивания (_). Названия модулей не должны содержать пробелов;

b. Первые 8 символов названия модуля должны быть уникальными. Заголовочные файлы должны иметь расширение .h, файлы исходного кода – .c;

c. Имя заголовочного файла не должно совпадать с именами заголовочных файлов стандартных библиотек языка C и C++ (например, *stdio* или *math*);

d. Название модуля, содержащего функцию *main()*, должно включать в себя слово «main».

Пример: см. приложение D.

Объяснение: использование кросс-платформенных сред разработки (например, с одновременной поддержкой Unix и Windows) является скорее нормой, чем исключением. Имена файлов со смесью верхнего и нижних регистров могут вызывать проблемы в разных операционных системах, а также приводить к ошибкам, потому что разработчики могут путать файлы с похожими именами, отличающимися только регистром символов. Включение слова «main» в имя файла помогает в процессе сопровождения кода – в частности, в проектах с несколькими конфигурациями ПО.

Применение: контроль соблюдения этих правил должен производиться в процессе автоматической проверки каждой сборки ПО.

4.2. Заголовочные файлы

Правила:

a. Каждому файлу исходного кода должен соответствовать ровно один заголовочный файл. Названия заголовочного файла и файла исходного кода должны совпадать;

b. Каждый заголовочный файл должен содержать защиту от повторного включения (см. пример)[9];

c. Заголовочный файл должен определять только процедуры (с помощью прототипов или макросов), константы (с помощью директивы препроцессора *#define*) и типы данных (с помощью ключевого слова *typedef*), которые будут использоваться в других модулях.

i. Рекомендуется избегать объявления переменных (с помощью ключевого слова *extern*) в заголовочном файле;

ii. В заголовочном файле не должно производиться выделения памяти для хранения переменных;

d. К публичному заголовочному файлу не должен быть подключен (с помощью директивы препроцессора *#include*) ни один приватный заголовочный файл.

Пример:

```
#ifndef ADC_H
#define ADC_H
...
#endif /* ADC_H */
```

Объяснение: стандарт языка C определяет, что по умолчанию все переменные и функции имеют глобальную область видимости. Это приводит к появлению ненужных (и опасных) связей между модулями. Чтобы снизить число лишних связей между модулями следует перенести как можно больше (в той мере, в какой это возможно) процедур, констант, типов данных и переменных в файл исходного кода.

См. также статью *Что входит в состав заголовочных файлов языка C?*

Применение: проверка выполнения этих правил должна производиться во время code review.

[9] Директива препроцессора #pragma once, выполняющая эту задачу, является непереносимой.

4.3. Файлы исходного кода

Правила:

a. Каждый файл исходного кода должен включать только логику управления одной «сущностью». Сущность может представлять собой инкапсулированный тип данных, активный объект, драйвер периферии (например, UART) или протокол (например, ARP);

b. Каждый файл исходного когда должен состоять из нескольких (или всех) приведенных ниже разделов, при этом порядок разделов должен соответствовать приведенному:

- блок комментариев;
- директивы препроцессора *#include*;
- объявление типов данных, констант и макросов;
- объявление переменных;
- прототипы приватных функций;
- реализация публичных функций;
- реализация приватных функций.

c. В каждом файле исходного кода всегда должен быть подключен (*#include*) заголовочный файл с тем же именем (например, к файлу *adc.c* должен быть подключен *adc.h*), чтобы позволить компилятору подтвердить, что каждая публичная функция соответствует ее прототипу;

d. В директиве препроцессора *#include* не должны использоваться абсолютные пути к файлам;

e. В каждом файле исходного кода не должно быть подключено неиспользуемых заголовочных файлов;

f. К файлу исходного кода не должен быть подключен другой файл исходного кода.

Пример: см. приложение D.

Объяснение: назначение и внутреннее устройства файла исходного кода должны быть понятны всем, кто занимается его поддержкой. Например, публичные функции, как правило, представляют наибольший интерес для разработчиков и поэтому разумно размещать их перед приватными функциями, которые они вызывают. Крайне важно, чтобы каждое объявление функции проверялось компилятором на соответствие прототипу.

Применение: большинство статических анализаторов позволяют определять неиспользуемые заголовочные файлы. Проверка выполнения остальных правил должна производиться во время code review.

4.4. Шаблоны файлов

Правила:

a. Для проекта должен быть создан набор шаблонов для заголовочных файлов и файлов исходного кода.

Пример: см. примеры шаблонов в приложениях B и C.

Объяснение: создание новых модулей на базе шаблонов обеспечивает согласованность блоков комментариев в начале файлов и гарантирует включение соответствующих уведомлений об авторских правах.

Применение: проверка выполнения этого правила должна производиться во время code review.

5. Типы данных

5.1. Названия

Правила:

a. Названия пользовательских типов данных (включая структуры, объединения и перечисления) должны включать в себя только символы нижнего регистра и символ подчеркивания (_), и заканчиваться постфиксом «_t»;

b. Структуры, объединения и перечисления должны быть объявлены с помощью ключевого слова *typedef*;

c. Названия публичных типов данных должны начинаться с префикса, состоящего из имени модуля и символа подчеркивания.

Пример:

```
typedef struct
{
    uint16_t count;
    uint16_t max_count;
    uint16_t _unused;
    uint16_t control;
} timer_reg_t;
```

Объяснение: названия типов данных и имена переменных часто похожи друг на друга. Например, набор регистров управления таймером в периферийном устройстве называется «timer_reg». В этом случае разумно создать структуры с названием «timer_reg_t», а «timer_reg» будет являться именем ее экземпляра. При необходимости эту же структуру можно использовать для создания теневой копии регистров таймера – например, в виде экземпляра с названием «timer_reg_shadow».

Применение: проверка использования ключевых слов *struct*, *union* и *enum* исключительно совместно с ключевым словом *typedef* или в анонимных объявлениях должна автоматически выполняться во время сборки. Проверка выполнения правил именования типов данных должна производиться во время code review.

5.2. Целочисленные типы фиксированного размера

Правила:

a. Каждый раз, когда размер целочисленного значения в битах или байтах играет роль для корректной работы программы, должен использоваться один из целочисленных типов данных фиксированного размера:

Размер	Знаковый тип	Беззнаковый тип
8 бит	int8_t	uint8_t
16 бит	int16_t	uint16_t
32 бита	int32_t	uint32_t
64 бита	int64_t	uint64_t

b. Ключевые слова *short* и *long* не должны использоваться в коде;

c. Ключевое слово *char* должно использоваться только при объявлениях и операциях со строками.

Пример: см. приложение D.

Объяснение: в стандарте C90 размер типов *char*, *short*, *int*, *long* и *long long* был осознанно сделан определяемым конкретной реализацией (implementation-defined), что привело к проблемам с переносимостью кода. Стандарт C99 не решил эту проблему, но добавил типы фиксированного размера, приведенные в таблице выше, которые определены в заголовочном файле *stdint.h*.

См. также статью Переносимые целочисленные типы фиксированного размера в языке C.

При отсутствии возможности использования компилятора, совместимого с C99, допустимо определить собственный набор типов фиксированного размера согласно приведенной выше таблице на основе базовых типов. Если это необходимо, обязательно используйте проверки во время компиляции (например, статические утверждения).

Применение: проверка отсутствия ключевых слов *short* и *long* должна автоматически выполняться во время сборки. Проверка выполнения остальных правил должна производиться во время code review.

5.3. Знаковые и беззнаковые целые числа

Правила:

a. Знаковые целочисленные типы не должны использоваться для создания битовых полей;

b. Знаковые целые числа не должны использоваться совместно с битовыми операторами (например, &, |, ~, ^, << и >>);

c. Знаковые целые числа не должны использоваться в операциях сравнения и других выражениях совместно с беззнаковыми целыми. Константы беззнаковых целых чисел должны объявляться с литералом «u».

Пример:

```
uint16_t unsigned_a = 6u;
int16_t signed_b = -9;

if (unsigned_a + signed_b < 4)
{
    // Это выражение кажется корректным, ведь -9 + 6 равно -3
    ...
}
// ... но компиляторы, в которых int является 16-битным, могут интерпретировать это...
// ...как (0xFFFF - 9) + 6
```

Объяснение: некоторые детали манипулирования бинарными данными, представленными в виде знаковых целых чисел, описаны в стандарте ISO C как «имеющие неуточненное поведение». Кроме того, совместное использование в выражениях целых чисел со знаком и без знака может привести к разным результатам на разных платформах[10]. Учтите, что целочисленные типы фиксированного размера, появившиеся в C99, сами по себе не предотвращают такие ошибок.

Применение: проверка выполнения этих правил должна выполняться статическим анализатором.

[10] См. [MISRA-C], приложения C и D.

5.4. Типы с плавающей точкой

Правила:

a. По возможности избегайте использования констант и переменных с плавающей точкой. В качестве альтернативы можно использовать числа с фиксированной точкой;

b. Если без чисел с плавающей точкой не обойтись:

i. Используйте типы, определенные в C99: float32_t, float64_t и float128_t.

ii. Используйте постфикс «f» для констант одинарной точности (например, pi = 3.141592f);

iii. Если вы используете числа двойной точности – убедитесь, что ваш компилятор их поддерживает;

iv. Никогда не проверяйте на равенство или неравенство значения с плавающей точкой;

v. Всегда вызывайте макрос *isfinite()* после вычислений, связанных со значениями с плавающей точкой, чтобы проверить, не является ли результат бесконечностью или NaN.

Пример:

```
#include <limits.h>
#if (DBL_DIG < 10) // Проверка, что компилятор поддерживает числа двойной точности.
# ошибка "Числа двойной точности не поддерживаются!"
#endif
```

Объяснение: значительное число потенциальных ошибок связано с некорректной работой с числами с плавающей точкой[11]. По умолчанию в языке C все константы с плавающей точкой представляются как числа двойной точности, что может быть неэффективным с точки зрения использования ресурсов или не поддерживаться на используемой платформе. Кроме того, многие микроконтроллеры не имеют аппаратной поддержки вычислений с плавающей точкой. Компилятор может не предупреждать об этом и выполнять математические операции над числами с плавающей точкой, подключая большую по размеру (обычно несколько килобайт кода) и медленную (множество инструкций на каждую операцию) библиотеку эмуляции поддержки вычислений с плавающей точкой.

Применение: проверка выполнения этих правил должна производиться во время code review.

[11] См. [CERT-C], п. 5.

5.5. Структуры и объединения

Правила:

a. Следует предпринять меры для отключения выравнивания памяти в структурах и объединениях, используемых при обмене с периферией или другими устройствами по внутренней шине или сети;

b. Следует предпринять меры для предотвращения изменения компилятором предполагаемого порядка битов[12] в битовых полях.

Пример:

```
typedef struct
{
    uint16_t count; // offset 0
    uint16_t max_count; // offset 2
    uint16_t _unused; // offset 4
    uint16_t enable : 2; // offset 6 bits 15-14
    uint16_t b_interrupt : 1; // offset 6 bit 13
    uint16_t _unused1 : 7; // offset 6 bits 12-6
    uint16_t b_complete : 1; // offset 6 bit 5
    uint16_t _unused2 : 4; // offset 6 bits 4-1
    uint16_t b_periodic : 1; // offset 6 bit 0
} timer_reg_t;

// Препроцессор проверят число байт, описывающих регистры таймера.
#if ((8 != sizeof(timer_reg_t))
# error "некорректный размер структуры timer_reg_t (ожидалось 8 байт)"
#endif
```

Объяснение: из-за различий между процессорами и размытостью формулировок в стандарте ISO C множество аспектов, связанных со структурами и объединениями, имеют неуточненное поведение, определяемое конкретной реализацией. В частности, существуют серьезные проблемы с переносимостью битовых полей, включая отсутствие стандартного порядка битов и официальной поддержки целочисленных типов фиксированного размера, совместно с которыми они обычно и используются. Для контроля размеров структур могут применяться статические утверждения или другие проверки, выполняемые на этапе компиляции, а также использоваться директивы препроцессора для выбора конкретного варианта структуры в зависимости от используемого компилятора.

Применение: проверка выполнения этих правил должна производиться во время code review.

[12] Вероятно, речь идет о порядке байт в битовых полях (прим. пер.)

5.6. Логический тип данных

Правила:

a. Логические переменные должны иметь тип *bool*.

b. Переменные других типов должны преобразовываться к булевским с помощью логических операторов (например, < или !=), но не через операторы преобразования типов.

Пример:

```
#include <stdbool.h>
...
bool b_in_motion = (0 != speed_in_mph);
```

Объяснение: стандарт C90 не определял тип данных для логических переменных, и разработчики обычно рассматривали любое ненулевое целочисленное значение как «истинное» (TRUE). Стандарт C99 позволяет использовать такой подход, но также включает в себя новый тип данных для логических переменных и константы *true* и *false*, объявленные в заголовочном файле *stdbool.h*.

Применение: проверка выполнения этих правил должна производиться во время code review.

6. Процедуры

6.1. Названия

Правила:

a. Имя процедуры не должно совпадать с ключевым словом, зарезервированным любым стандартом языка C или C++. К ключевым словам относятся *interrupt*, *inline*, *class*, *true*, *false*, *public*, *private*, *friend*, *protected* и многие другие;

b. Имя процедуры не должно совпадать с именем функции из стандартной библиотеки языка C (например, *strlen*, *atoi* и *memset*);

c. Имя процедуры не должно начинаться с символа подчеркивания (_);

d. Имя процедуры не должно быть длиннее 31 символа;

e. Имя функции не должно включать в себя символы верхнего регистра;

f. Имя макроса не должно включать в себя символы нижнего регистра;

g. Если имя процедуры состоит из нескольких отдельных слов, то они должны разделяться символами подчеркивания (_);

h. Имя процедуры должно описывать ее назначение. Так как процедуры инкапсулируют «действия» программы, то разумно использовать в их именах глаголы (например, adc_read(); рекомендуется использовать связку «существительное-глагол»). Альтернативный вариант – описать в имени процедуры результат ее выполнения (например, led_is_on());

i. Имена всех публичных функций должны начинаться с имени их модуля и символа подчеркивания (например, sensor_read()).

Пример: см. приложение D.

Объяснение: хорошо выбранные имена функций упрощают анализ и поддержку кода (и, следовательно, снижают затраты на его поддержку). Данные (переменные) программ являются «существительными». Функции манипулируют данными и, таким образом, являются «глаголами». Использование префиксов модулей обеспечивает инкапсуляцию и помогает избежать дублирования имен процедур.

Применение: выбор имен осуществляется на этапе проектирования ПО, а проверка выполнения остальных правил должна производиться во время code review.

6.2. Функции

Правила:

a. Следует приложить все возможные усилия, чтобы код функции помещался на одну печатную страницу и не превышал 100 строк;

b. По возможности все функции должны начинаться с новой печатной страницы, за исключением тех случаев, когда несколько функций помещаются на одной странице[13];

c. Рекомендуется создавать функции с единственной точкой выхода, при этом оператор *return* должен быть размещен в последних строках функции;

d. Для каждой публичной функции в заголовочном файле ее модуля должен быть объявлен прототип;

e. Все приватные функции должны быть объявлены как статические;

f. Каждый параметр функции должен быть объявлен явным образом и иметь осмысленное имя.

Пример:

```
int
state_change (int event)
{
    int result = ERROR;
    if (EVENT_A == event)
    {
        result = STATE_A;
    }
    else
    {
        result = STATE_B;
    }
    return (result);
}
```

Объяснение: code review часто проводится путем чтения распечаток программ. Поэтому в идеале каждая функция должна помещаться на одной печатной странице, чтобы перелистывание страниц не отвлекало рецензентов.

Несколько операторов *return* в рамках одной функции следует использовать только тогда, когда это улучшает читабельность кода.

Применение: проверка выполнения этих правил должна производиться во время code review.

[13] Один из способов добиться этого — вставить символ перевода страницы «FF» (ASCII-код 0x0C) в начале первой строки блока комментариев перед объявлением функции.

6.3. Макросы в стиле функций

Правила:

a. Макросы с аргументами должны использовать только в тех случаях, когда эту же задачу невозможно решить с помощью функции;

b. В случае использования макросов с аргументами руководствуйтесь следующими правилами:

i. Тело макроса должно быть заключено в круглые скобки;

ii. Каждый параметр должен быть заключен в круглые скобки;

iii. Каждый параметр должен использоваться не более одного раза, чтобы избежать побочных эффектов;

iv. Никогда не производите в макросе изменение потока выполнения (например, не включайте в макрос оператор *return*).

Пример:

```
// Не делайте так...
#define MAX(A, B) ((A) > (B) ? (A) : (B))
// ...если можно сделать вот так.
inline int max(int num1, int num2)
```

Объяснение: существует множество рисков, связанных с использованием директив препроцессора, и многие из них касаются макросов с аргументами. Явное использование круглых скобок (как показано в примере), безусловно, важно, но оно не устраняет вероятность случайного двойного вызова макроса (например, в такой форме: MAX(i++, j++)). Если в макрос вместо значения передается выражение и при этом в самом макросе аргументы не заключены в скобки, то после подстановки выражение может быть изменено, например, из-за приоритета операций.

Другие потенциальные ошибки, связанные с некорректным использованием макросов, касаются сравнения знаковых и беззнаковых чисел, а также проверок на равенство и неравенство значений с плавающей запятой. Что еще хуже, макросы «невидимы» в рантайме, и, следовательно, их невозможно запустить в отладчике.

Если в вашей задаче важна производительность, то обратите внимание на ключевое слово *inline*, добавленное в стандарте C99 (и позаимствованное из C++).

Применение: проверка выполнения этих правил должна производиться во время code review.

6.4. Потоки выполнения

Правила:

а. Имена функций, которые инкапсулируют потоки выполнения (также называемые «задачами» или «процессами»), должны иметь постфикс «_thread» (или «_task», «_process»).

Пример:

```
void
alarm_thread (void * p_data)
{
    alarm_t alarm = ALARM_NONE;
    int err = OS_NO_ERR;
    for (;;)
    {
        alarm = OSMboxPend(alarm_mbox, &err);
        // Здесь будет обработка тревог.
    }
}
```

Объяснение: каждая задача в операционной системе реального времени (RTOS) похожа на мини-вариант функции main() и обычно выполняется в бесконечном цикле. Необходимо иметь возможность легко найти эти важные асинхронные функции во время проверки кода и сеансов отладки.

Применение: выбор имен осуществляется на этапе проектирования ПО, а проверка выполнения этого правила должна производиться во время code review.

6.5. Обработчики прерываний

Правила:

a. Обработчики прерываний (Interrupt service routine, ISR) отличаются от обычных функций. Компилятор должен знать о том, что функция является ISR – для этого следует использовать директиву препроцессора *#pragma* или специальное ключевое слово конкретного компилятора (например, «*__interrupt*»);

b. Имена всех функций-обработчиков прерываний должны иметь постфикс «_isr»;

c. Чтобы гарантировать, что ISR не будут непреднамеренно вызываться из других фрагментов ПО (они могут нарушить работу CPU и стека вызовов, если это произойдет), каждая функция ISR должна быть объявлена как статическая и/или размещена в конце модуля соответствующего драйвера, если это допустимо на конкретной платформе;

d. «Заглушка» или ISR по умолчанию должны быть установлены в таблице векторов прерываний для всех непредвиденных или необрабатываемых источников прерываний. Каждая такая заглушка может попытаться запретить будущие прерывания того же типа, сообщить об этом контроллеру прерываний и вызвать и *assert()*.

Пример:

```
#pragma irq_entry
void
timer_isr (void)
{
    uint8_t static prev = 0x00; // предыдущее состояние кнопок
    uint8_t curr = *gp_button_reg; // текущее состояния кнопок
    // Сравниваем текущее и предыдущее состояние.
    g_debounced |= (prev & curr);
    g_debounced &= (prev | curr);
    // Сохраняем текущее состояние пинов для следующего прерывания.
    prev = curr;
    // При необходимости подтвердите прерывание таймера на аппаратном уровне.
}
```

Объяснение: ISR напрямую связаны с аппаратным обеспечением. По определению они и остальной код выполняются асинхронно. Если они совместно используют глобальные переменные или регистры, то эти объекты должны быть защищены с помощью временного отключения прерываний при обработке их кодом программы. ISR не должен «зависать» внутри операционной системы или ждать изменения значения переменной или регистра.

Обратите внимание, что работа с ISR может отличаться на разных платформах – например, может потребоваться, чтобы функции ISR имели прототипы и были «видны» хотя бы одной функции.

Хотя «заглушки»» для неиспользуемых прерываний напрямую не предотвращают ошибки в ПО, они определенно могут сделать систему более надежной для применения в «полевых» условиях.

Применение: проверка выполнения этих правил должна производиться во время code review.

7. Переменные

7.1. Именование

Правила:

a. Имя переменной не должно совпадать с ключевым словом, зарезервированным любым стандартом языка C и C++ (особенно стандартами K&R и C99). К ключевым словам относятся *interrupt*, *inline*, *class*, *true*, *false*, *public*, *private*, *friend*, *protected* и многие другие;

b. Имя переменной не должно совпадать с именем переменной из стандартной библиотеки языка C (например, *errno*);

c. Имя переменной не должно начинаться с символа подчеркивания (_);

d. Имя переменной не должно быть длиннее 31 символа;

e. Имя переменной не должно быть короче 3 символов (включая счетчики циклов);

f. Имя переменной не должно включать в себя символы верхнего регистра;

g. Имя переменной не должно включать в себя числа, используемые где-то еще – например, размер массива или размер типа данных;

h. Если имя переменной состоит из нескольких отдельных слов, то они должны разделяться символами подчеркивания (_);

i. Имя переменной должно описывать ее назначение;

j. Имена глобальных переменных должны начинаться с префикса g_ (пример: g_zero_offset);

k. Имена указателей должны начинаться с префикса p_ (пример: p_led_reg);

l. Имена указателей на указатели должны начинаться с префикса pp_ (пример: pp_vector_table);

m. Имена булевских переменных и целочисленных переменных, используемых для хранения логических значений, должны начинаться с префикса b_ и формулироваться в виде ответов на вопросы (пример: b_done_yet, b_is_buffer_full);

n. Имена переменных, используемых для хранения дескрипторов (например, файловых дескрипторов) должны начинаться с префикса h_ (пример: h_input_file);

o. Если имя переменной подразумевает наличие нескольких префиксов, то их порядок должен быть следующим: [g][p|pp][b|h].

Пример: см. приложение D.

Объяснение: основные из перечисленных правил нужны для обеспечения максимальной переносимости кода между компиляторами. Многие компиляторы C распознают различия только первых 31 символов имени переменной и резервируют имена, начинающиеся с символа подчеркивания, для системных переменных.

Другие правила предназначены для уменьшения вероятности некорректного использования переменных. Например, весь код, связанный с использованием глобальных переменных и других глобальных объектов, включая регистры периферии, должен быть тщательно проработан, чтобы гарантировать отсутствие состояния гонки или повреждения данных при асинхронной записи.

Применение: проверка выполнения этих правил должна производиться во время code review.

7.2. Инициализация

Правила:

a. Все переменные должны быть инициализированы перед использованием в коде;

b. Рекомендуется объявлять локальные переменные непосредственно перед их использованием, а не все сразу в начале функции;

c. Если используются глобальные переменные файла или проекта – то они должны быть объявлены все вместе в начале файла исходного кода;

d. Каждая переменная-указатель, не имеющая начального адреса, должна быть инициализирована значением NULL.

Пример:

```
uint32_t g_array[NUM_ROWS][NUM_COLS] = { ... };
...
    for (int col = 0; col < NUM_COLS; col++)
    {
        g_array[row][col] = ...;
    }
```

Объяснение: многие разработчики предполагают, что некоторые вещи будут выполняться в рантайме автоматически – например, обнуление значений неинициализированных переменных при запуске системы. Это неверное предположение, которое может оказаться опасным в критически важной системе. Для повышения читабельности кода лучше объявлять локальные переменные как можно ближе к моменту их первого использования[14] – этот механизм был поддержан в стандарте C99 за счет включения в него соответствующей возможности C++.

Применение: проверка на использование в коде неинициализированных переменных должна автоматически выполняться статическим анализатором во время сборки ПО. Проверка выполнения остальных правил должна производиться во время code review.

[14] [Uwano] приводит описание движений глаз при пролистывании кода вверх и вниз в процессе code review, которые показывают ценность размещения объявлений переменных как можно ближе к месту их использования.

8. Операторы

8.1. Объявление переменные

Правила:

а. Не объявляйте переменные через запятую.

Пример:

```
char * x, y; // Подразумевалось, что y тоже будет указателем? Не делайте так.
```

Объяснение: объявление каждой переменной в отдельной строке не вызывает проблем. С другой стороны, при объявлении переменных через запятую велика вероятность того, что компилятор или программист, который будет сопровождать код, поймут вас неправильно.

Применение: проверка выполнения этого правила должна производиться во время code review.

8.2. Оператор if

Правила:

a. Рекомендуется в первых проверках операторов *if* и *else if* размещать самые короткие условия;

b. Операторы *if…else* не должны иметь более двух уровней вложенности. Используйте вызовы функций или оператор *switch*, чтобы сделать код более понятным;

c. В проверках условий операторов *if* и else *if* не должен использоваться оператор присваивания (=);

d. Каждый оператор *if*, включающий в себя один или несколько операторов *else if*, должен заканчиваться оператором *else*[15].

Пример:

```
if (NULL == p_object)
{
    result = ERR_NULL_PTR;
}
else if (p_object = malloc(sizeof(object_t))) // Так делать нельзя!
{
    ...
}
else
{
    // Много строк кода для типичного случая выполнения
    ...
}
```

Объяснение: длинные условия затрудняют понимание логики принятия решения. Размещая короткие условия первыми, вы облегчаете ее понимание (а это, в свою очередь, уменьшает число ошибок). Много уровней вложенности операторов *if…else* обычно является признаком сложной и «хрупкой» реализации конечного автомата; всегда есть более безопасный и читаемый способ для реализации такой же логики.

Применение: проверка выполнения этих правил должна производиться во время code review.

[15] Аналогично оператор *switch* должен заканчиваться меткой *default*.

8.3. Оператор switch

Правила:

a. Оператор *break* должен выравниваться по метке *case* (а не коду метки);

b. Каждый оператор *switch* должен включать в себя метку *default*;

c. Если один шаг допускает переход на другой (то есть не заканчивается оператором *break*), то это должно быть явным образом прокомментировано.

Пример:

```
switch (err)
{
    case ERR_A:
    ...
    break;

    case ERR_B:
    ...
    // Также выполняем код метки ERR_C.

    case ERR_C:
    ...
    break;

    default:
    ...
    break;
}
```

Объяснение: оператор *switch* представляет собой мощный инструмент, но его использование часто сопряжено с ошибками – например, пропущенными операторами *break* и забытыми метками. Выравнивание операторов *break* по меткам повышает шансы увидеть такие ошибки.

Применение: проверка выполнения этих правил должна производиться во время code review.

8.4. Циклы

Правила:

a. В качестве начального и конечного значения счетчика цикла не должны использоваться «магические числа»[16];

b. Значение счетчика цикла оператора for должно изменяться только в первом (инициализация) и третьем выражении самого оператора, но не в теле цикла и за его пределами;

c. Для организации бесконечного цикла следует использовать выражение (;;)[17];

d. Каждый цикл, тело которого не включает кода, должен содержать комментарий, поясняющий, почему не нужно ничего делать до завершения этого цикла.

Пример:

```
// Зачем использовать магические числа (например, 100) в своем коде?
for (int row = 0; row < 100; row++)
{
    // Использование констант с ясными именами уменьшает число ошибок...
    // ...и повышает читабельность.
    for (int col = 0; col < NUM_COLS; col++)
    {
    ...
    }
...
}
```

Объяснение: всегда важно синхронизировать количество итераций цикла с размером обрабатываемой структуры данных. Использование констант в качестве верхнего значения счетчика цикла предотвращает ошибки, которые возникают в тот момент, когда изменения в одном фрагменте кода (например, размерности массива) не вносятся в других его фрагментах.

Применение: проверка выполнения этих правил должна производиться во время code review.

[16] Обратите внимание, что оператор *sizeof* в теории удобен для вычисления верхней границы счетчика цикла, но этот метод не работает, если вы передаете указатель на массив вместо самого массива.

[17] Керниган и Ритчи давно рекомендовали этот способ, указывая, что он имеет дополнительное преимущество перед *while(1)*, который в некоторых случаях (при использовании определенных шрифтов) можно спутать с *while(l)*.

8.5. Переходы

Правила:

a. Как уже было указано в правиле 1.7.c – рекомендуется не использовать оператор *goto*. Если этот оператор всё же используется – то он должен применяться только для перехода к метке, расположенной в этом же или соседнем блоке кода;

b. Следующие функции стандартной библиотеки языка С не должны использоваться в коде программы: *abort()*, *exit()*, *setjmp()* и *longjmp()*.

Объяснение: алгоритмы, использующие переходы для изменения потока управления, могут и должны быть переписаны таким образом, чтобы стать более читабельными и, следовательно, более простыми в сопровождении.

Применение: проверка наличия в коде запрещенных операторов должна автоматически производиться на этапе сборки. В случае использования *goto* – во время code review должна быть выполнена оценка целесообразности этого шага в данном конкретном случае с изучением возможных альтернативных вариантов.

8.6. Проверка на равенство

Правила:

а. При проверке переменной на равенство константе – константа должна располагаться слева от оператора проверки (==).

Пример:

```
if (NULL == p_object)
{
    return (ERR_NULL_PTR);
}
```

Объяснение: желательно обнаруживать возможные опечатки и другие ошибки еще на этапе компиляции. На этапе эксплуатации ошибки уже сложнее обнаружить и труднее устранить. Данное правило сформулировано из приведенных выше соображений – любой компилятор гарантированно обнаружит ошибочные попытки присваивания (т.е. = вместо ==) значения переменной константе.

Применение: многие компиляторы можно настроить таким образом, чтобы они предупреждали о подозрительных присваиваниях (т.е. об использовании оператора присваивания там, где можно ожидать оператор сравнения). Однако окончательная ответственность за соблюдения этого правила остается за рецензентами кода.

Приложение А. Список распространенных сокращений

Следующие сокращения и аббревиатуры могут использоваться в исходном коде без каких-либо пояснений.

Сокращение	Полный термин	Перевод
adc	analog-to-digital converter	аналого-цифровой преобразователь
avg	average	среднее
b_	Boolean	префикс логического (булевского) значения
buf	buffer	буфер
cfg	configuration	конфигурация
curr	current (item in a list)	текущий объект в списке
dac	digital-to-analog converter	цифро-аналоговый преобразователь
ee	EEPROM	EEPROM
err	error	ошибка
g_	global	префикс глобальной переменной
gpio	general-purpose I/O pins	интерфейс ввода/вывода общего назначения
h_	handle (to)	дескриптор
init	initialize	инициализация
io	input/output	ввод-вывод
isr	interrupt service routine	обработчик прерываний
lcd	liquid crystal display	жидкокристаллический дисплей
led	light-emitting diode	светодиод
max	maximum	максимум
mbox	mailbox	«почтовый ящик»
mgr	manager	менеджер объектов
min	minimum	минимум
msec[18]	millisecond	миллисекунды
msg	message	сообщение
next	next (item in a list)	следующий объект в списке
nsec	nanosecond	наносекунды
num	number (of)	количество
p_	pointer (to)	префикс указателя
pp_	pointer to a pointer (to)	префикс указателя на указатель
prev	previous (item in a list)	предыдущий объект в списке
prio	priority	приоритет
pwm	pulse width modulation	широтно-импульсная модуляция
q	queue	очередь
reg	register	регистр
rx	receive	приём данных
sem	semaphore	семафор
str	string (null terminated)	нуль-терминированная строка
sync	synchronize	синхронизация
temp	temperature	температура
tmp	temporary	временный объект
tx	transmit	передача данных
usec	microsecond	микросекунды

[18] Обратите внимание, что секунды, минуты, часы, дни, недели, месяцы и года не должны сокращаться. Это, в том числе, устраняет конфликт между «минутами» и «минимумом» (минуты – minute, минимум – min).

Приложение B. Шаблон заголовочного файла

```
/** @file module.h
*
* @brief Информация о назначении данного модуля.
*
* @par
* COPYRIGHT NOTICE: (c) 2018 Barr Group. All rights reserved.
*/

#ifndef MODULE_H
#define MODULE_H

int8_t max8(int8_t num1, int8_t num2);

#endif /* MODULE_H */

/*** end of file ***/
```

Приложение C. Шаблон файла исходного кода

```
/** @file module.c
*
* @brief Информация о назначении данного модуля.
*
* @par
* COPYRIGHT NOTICE: (c) 2018 Barr Group. All rights reserved.
*/

#include <stdint.h>
#include <stdbool.h>

#include “module.h”

/*!
* @brief Определение наибольшего из двух 8-битных целых числе.
*
* @param[in] num1 Первое сравниваемое число.
* @param[in] num2 Второе сравниваемое число.
*
* @return Большее из двух чисел.
*/
int8_t
max8 (int8_t num1, int8_t num2)
{
    return ((num1 > num2) ? num1 : num2);
}

/*** end of file ***/
```

Приложение D. Пример программы

```
/** @file crc.h
*
* @brief Компактная библиотека расчета CRC-CCITT, CRC-16 и CRC-32 для встраиваемых
* систем.
*
* @par
* COPYRIGHT NOTICE: (c) 2000, 2018 Michael Barr. This software is placed in the
* public domain and may be used for any purpose. However, this notice must not
* be changed or removed. No warranty is expressed or implied by the publication
* or distribution of this source code.
*/

#ifndef CRC_H
#define CRC_H

// Используемый алгоритм выбирается на этапе компиляции.
//
#if defined(CRC_CCITT)

#define CRC_NAME "CRC-CCITT"
typedef uint16_t crc_t;

#elif defined(CRC_16)

#define CRC_NAME "CRC-16"
typedef uint16_t crc_t;

#elif defined(CRC_32)

#define CRC_NAME "CRC-32"
typedef uint32_t crc_t;

#else

#error "Отсутствует #define для CRC_CCITT, CRC_16 или CRC_32”

#endif

// Публичные функции, предоставляемые библиотекой.
//
void crc_init(void);
crc_t crc_slow(uint8_t const * const p_message, int n_bytes);
crc_t crc_fast(uint8_t const * const p_message, int n_bytes);

#endif /* CRC_H */

/*** end of file ***/
```

```
/** @file crc.c
*
* @brief Компактный генератор CRC для встраиваемых систем с поддержкой алгоритма
* «грубой силы» и табличного алгоритма. Поддерживает стандарты CRC-CCITT, CRC-16 и
* CRC-32.
* @par
* COPYRIGHT NOTICE: (c) 2000, 2018 Michael Barr. This software is placed in the
* public domain and may be used for any purpose. However, this notice must not
* be changed or removed. No warranty is expressed or implied by the publication
* or distribution of this source code.
*/

#include <stdint.h>

#include "crc.h"

// Параметры алгоритма, выбранного в crc.h.
//
#define BITS_PER_BYTE 8
#define WIDTH (BITS_PER_BYTE * sizeof(crc_t))
#define TOPBIT (1 << (WIDTH - 1))

// Выделение памяти для таблицы поиска CRC.
//
#define CRC_TABLE_SIZE 256
static crc_t g_crc_table[CRC_TABLE_SIZE];

// Дальнейшая конфигурация алгоритма для поддержки выбранного стандарта CRC.
//
#if defined(CRC_CCITT)

#define POLYNOMIAL ((crc_t) 0x1021)
#define INITIAL_REMAINDER ((crc_t) 0xFFFF)
#define FINAL_XOR_VALUE ((crc_t) 0x0000)
#define REFLECT_DATA(X) (X)
#define REFLECT_REMAINDER(X) (X)

#elif defined(CRC_16)

#define POLYNOMIAL ((crc_t) 0x8005)
#define INITIAL_REMAINDER ((crc_t) 0x0000)
#define FINAL_XOR_VALUE ((crc_t) 0x0000)
#define REFLECT_DATA(X) ((uint8_t) reflect((X), BITS_PER_BYTE))
#define REFLECT_REMAINDER(X) ((crc_t) reflect((X), WIDTH))

#elif defined(CRC_32)

#define POLYNOMIAL ((crc_t) 0x04C11DB7)
#define INITIAL_REMAINDER ((crc_t) 0xFFFFFFFF)
#define FINAL_XOR_VALUE ((crc_t) 0xFFFFFFFF)
#define REFLECT_DATA(X) ((uint8_t) reflect((X), BITS_PER_BYTE))
#define REFLECT_REMAINDER(X) ((crc_t) reflect((X), WIDTH))

#endif
```

```
/*!
* @brief Получение «отражения» для набора бит.
* @param[in] data Набор бит.
* @param[in] num2 Число бит.
* @return Отраженный набор бит.
*/
static uint32_t
reflect (uint32_t data, uint8_t n_bits)
{
    uint32_t reflection = 0x00000000;

    // ПРИМЕЧАНИЕ. Для повышения эффективности не проверяется, что n_bits <= 32.
    //
   for (uint8_t bit = 0; bit < n_bits; ++bit)
   {
       // Если младший бит установлен, то установим его отражение.
       //
       if (data & 0x01)
       {
           reflection |= (1 << ((n_bits - 1) - bit));
       }

       data = (data >> 1);
   }

return (reflection);

} /* reflect() */

/*!
* @brief Инициализация таблицы поиска для ускорения побайтового вычисления CRC.
*
* @par
* Эта функция должна вызываться перед crc_fast() или перед созданием таблицы в ROM.
*/
void
crc_init (void)
{
    // Вычисляем остаток для каждого возможного делимого.
    //
    for (crc_t dividend = 0; dividend < CRC_TABLE_SIZE; ++dividend)
    {
        // Для начала дополним делимое нулями.
        //
        crc_t remainder = dividend << (WIDTH - BITS_PER_BYTE);
        // Выполняем деление по модулю 2, по одному биту за раз.
        //
        for (int bit = BITS_PER_BYTE; bit > 0; --bit)
        {
            // Пытаемся делить текущий бит данных.
            //
            if (remainder & TOPBIT)
            {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            }
            else
            {
                remainder = (remainder << 1);
            }
         }

    // Сохраним результаты в таблицу.
    //
    g_crc_table[dividend] = remainder;
    }
} /* crc_init() */
```

```
/*!
* @brief Функция рассчитывает CRC для массива байт, бит за битом.
* @param[in] p_message Указатель на массив байт, для которого будет рассчитано CRC.
* @param[in] n_bytes Число байт в массиве.
* @return Рассчитанная CRC.
*/

crc_t
crc_slow (uint8_t const * const p_message, int n_bytes)
{
    crc_t remainder = INITIAL_REMAINDER;
    // Выполняем деление по модулю 2, по одному байту за раз.
    //
    for (int byte = 0; byte < n_bytes; ++byte)
    {
        // Переходим к остатку деления предыдущего байта.
        //
        remainder ^= (REFLECT_DATA(p_message[byte]) << (WIDTH - BITS_PER_BYTE));
        // Выполняем деление по модулю 2, по одному биту за раз.
        //
        for (int bit = BITS_PER_BYTE; bit > 0; --bit)
        {
        // Пытаемся делить текущий бит данных.
        //
            if (remainder & TOPBIT)
            {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }
    }
    // Остатком от последнего деления является CRC.
    //
    return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);

} /* crc_slow() */
```

```
/*!
* @brief Функция рассчитывает CRC для массива байт, байт за байтом.
* @param[in] p_message Указатель на массив байт, для которого будет рассчитано CRC.
* @param[in] n_bytes Число байт в массиве.
* @return Рассчитанная CRC.
*/

crc_t
crc_fast (uint8_t const * const p_message, int n_bytes)
{
    crc_t remainder = INITIAL_REMAINDER;
    // побайтово делим сообщение на полином
    //
    for (int byte = 0; byte < n_bytes; ++byte)
    {
        uint8_t data = REFLECT_DATA(p_message[byte]) ^
            (remainder >> (WIDTH - BITS_PER_BYTE));
        remainder = g_crc_table[data] ^ (remainder << BITS_PER_BYTE);
    }

    // Остатком от последнего деления является CRC.
    //
    return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);
} /* crc_fast() */

/*** end of file ***/
```

Список литературы

[Barr] Barr, Michael. “Programming Embedded Systems in C and C++.” O’Reilly, 1999.

[C90] “ISO/IEC9899:1990, Programming Languages – C,” ISO, 1990.

[C99] “ISO/IEC9899:1999, Programming Languages – C,” ISO, 1999.

[CERT-C] Seacord, Robert C. “The CERT C Coding Standard, Second Edition.” Pearson, 2014.

[Harbison] Harbison III, Samuel P. and Guy L. Steele, Jr. “C: A Reference Manual, Fifth Edition.” Prentice Hall, 2002.

[Hatton] Hatton, Les. “Safer C: Developing Software for High-Integrity and SafetyCritical Systems.” McGraw-Hill, 1994.

[Holub] Holub, Allen I. “Enough Rope to Shoot Yourself in the Foot: Rules for C and C++ Programming.” McGraw-Hill, 1995.

[IEC61508] “Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems,” International Electromechanical Commission, 1998-2000.

[Koenig] Koenig, Andrew. “C Traps and Pitfalls.” Addison-Wesley, 1988.

[Loudon] Loudon, Kyle. “C++ Pocket Reference.” O’Reilly, 2003.

[MISRA-C] “MISRA C:2012 Guidelines for the use of the C language in critical systems,” MIRA, March 2013.

[MISRA-C++] “MISRA C++ Guidelines for the use of the C++ language in critical systems,” MIRA, June 2008.

[Prinz] Prinz, Peter and Ulla Kirch-Prinz. “C Pocket Reference.” O’Reilly, 2003.

[Sutter] Sutter, Herb and Andrei Alexandrescu. “C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.” Pearson, 2005.

[Uwano] Uwano, H., Nakamura, M., Monden, A., and Matsumoto, K. “Analyzing Individual Performance of Source Code Review Using Reviewer’s Eye Movement,” *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, San Diego, March 27-29, 2006.

“Стандарт кодирования на языке C для встраиваемых систем от Barr Group стал основой, использованной Pole/Zero для разработки собственного стандарт кодирования для встраиваемого ПО.”

Кевин Эхлерт, инженер-схемотехник, Pole/Zero

“Обучение и консультации Barr Group по их стандарту кодирования и MISRA C сыграли важную роль, поскольку Datalight разрабатывает файловую систему Reliance Edge, которая используется в критически важных с точки зрения безопасности системах.”

Тони Квестад, директор по разработке, Datalight

“Стандарт кодирования на языке C для встраиваемых систем значительно сэкономил нам время на разработку нашего собственного стандарта кодирования для уменьшения числа ошибок.»"

Эд Лукас, менеджер по электротехническому оборудованию, Chamberlain

Стандарт кодирования на языке C для встраиваемых систем от Barr Group был создан, чтобы помочь инженерам-программистам свести к минимуму число ошибок во встраиваемом ПО. В отличие от других стандартов кодирования, данный стандарт фокусируется на практических аспектах, предотвращающих появление ошибок, а также направленных на упрощение поддержки и повышения уровня переносимости встраимаего ПО. Стандарт включает в себя набор правил, а также специальные соглашения об именах объектов и другие правила использования типов данных, функций, макросов препроцессора, переменных и операторов. Отдельные правила, от следования которым объективно ожидается снижение числа ошибок, помечены особой иконкой.

Стандарт *BARR-C* отличается, но совместим со стандартом *MISRA C Guidelines for Use of the C Language in Critical Systems*. Разработчики могут с легкостью объединить правила этих стандартов.

Об авторе

Майкл Барр — технический директор и соучредитель Barr Group. Он является бывшим заместителем профессора электротехники и вычислительной техники с практическим опытом разработки и внедрения ПО в различных отраслях. Получив международное признание как эксперт в области архитектуры встроенного ПО и процессов разработки, Барр выступал в качестве эксперта в судах США и Канады. Барр также является автором книг Programming Embedded Systems in C и C++ и Embedded Systems Dictionary, а также более семидесяти статей и докладов о проектировании встраиваемых систем. Он был главным редактором журнала Embedded Systems Programming, а также членом консультативного совета и председателем конференции по встраиваемым системам. Барр получил степени бакалавра и магистра в области электротехники в Университете Мэриленда.

О Barr Group

Barr Group, The Embedded Systems Experts®, является независимым поставщиком инженерно-консультационных и обучающих услуг для отрасли встраиваемых систем. Barr Group разработала и опубликовала этот стандарт кодирования для уменьшения количества ошибок во встраиваемом ПО, чтобы помочь разработчикам повысить общую надежность и безопасность своих систем.

barrgroup.com

ISBN 9781721127986

Copyright © 2018 by Barr Group