# Некоторые мысли по поводу программирования встроенных систем

**Владимир ТАТАРЧЕВСКИЙ**
arktur04@mail.ru

**С огромным интересом прочитал я статью Иосифа Кашенбойма «Квадрига Аполлона и микропроцессоры». Многие проблемы, описанные в статье, знакомы мне по собственному опыту, по опыту разработки программного обеспечения микроконтроллерных систем, и мне хотелось бы поделиться с читателями журнала «Компоненты и технологии» некоторыми соображениями по данному поводу. Хочу подчеркнуть, что все нижеизложенное является моим личным мнением и ни в коем случае не претендует на истину в последней инстанции.**

## Дурная спираль

В современных рыночных условиях первоочередной задачей фирмы, занимающейся разработками, является наиболее быстрый вывод изделия на рынок. Казалось бы, для этого есть все условия: благо современная элементная база сводит процесс разработки практически к сборке схемы по принципу «детского конструктора». Микроконтроллеры, периферия, источники питания, всевозможные интерфейсы, любые датчики, индикаторы от старых добрых семисегментных до красивых графических ЖК-панелей — ассортимент выпускаемых компонентов огромен. В 1980-х годах разработчикам такие возможности и не снились. Вроде бы, бери да делай, были бы идеи и заказчики. И тем не менее сроки разработки зачастую затягиваются до бесконечности. Причины стары как мир (программного обеспечения): ошибки в программе и неправильно сформулированное/понятое техническое задание. Процесс разработки напоминает спираль: написали код, протестировали, выявили ошибки, исправили, протестировали, выявили несоответствие техзаданию, переписали код, протестировали, исправили, заказчик захотел что-нибудь переделать, снова переписали код… Спираль разворачивается, код становится все больше, заказчик хочет знать, когда же наконец проект будет закончен, разработчик хочет знать, сколько еще переделывать программу по милости заказчика. Знакомая ситуация, не правда ли?

## Производительность программиста не является константой

Каждый программист знает, сколько строк кода он может написать за рабочий день. Кто-то может выдать на-гора двести строк, кто-то — пятьсот, кто-то — тысячу. Однако если взять полный срок выполнения проекта и его объем в строках кода, общая производительность будет, как правило, гораздо ниже, чем потенциальные способности программиста. Каковы причины столь прискорбного положения вещей? В статье [1] в качестве основного фактора, влияющего на производительность работы программиста, названа степень комфортности его рабочего места и общих условий труда. С этим никто не спорит, каждый программист хочет работать в тишине и покое, не отвлекаясь на посторонние вещи и полностью сосредоточившись на коде. Только ли этот фактор влияет на производительность программиста? Отнюдь нет. В той же статье существенное внимание уделено «эффекту большой системы»: при росте проекта и количества занятых в нем разработчиков производительность каждого разработчика падает. В качестве причины такого явления обычно называют то, что часть (причем очень существенная) времени тратится на взаимодействия разработчиков, так как они вынуждены стыковать между собой свои части проекта. Естественно, значительные изменения в одной части проекта могут привести к переделке остальных частей, сведя на нет усилия всей команды. Объем информации, циркулирующей между разработчиками, растет как факториал их числа. В один прекрасный момент работа останавливается.

Глубокий анализ причин низкой производительности программистов приведен в [2, 3]. Однако, как пишет Брукс в своей замечательной статье «Серебряной пули нет» [3], не существует способа радикального повышения эффективности труда разработчиков.

## Для чего нужны суперпрограммисты?

В статье Кашенбойма предлагается метод «суперпрограммиста», предназначенный для решения проблем. Суть его заключается в том, что лучшим разработчикам фирмы даются маленькие кусочки проекта, и эти суперпрограммисты выполняют маленькие части проекта быстро и качественно. Но здесь могут возникнуть возражения. Во-первых, кто и как будет решать, какой маленький кусок наиболее важен для проекта в целом? Во-вторых, допустим, что суперпрограммист сделал свою небольшую часть. Однако проект-то еще не завершен. Другие программисты (не «супер») работают на своих участках. Можно, конечно, дать «суперу» еще одну часть проекта, но в целом сроки выполнения определяются совокупной производительностью всех участников проекта, а не производительностью суперпрограммиста. Да и проблема стыковки элементов проекта осталась. Далее, суперпрограммист наверняка попросит у вас суперзарплату. Суперпрограммист всегда найдет себе другую работу, а вот вы другого суперпрограммиста можете и не найти, если, конечно, у вас в фирме не суперзарплаты. В любом случае, суперпрограммист сделает все возможное, чтобы стать незаменимым. После его ухода из фирмы у вас останутся огромные исходники, а вот вся реальная информация о том, как все это взаимодействует, уйдет вместе с ним. В таком случае проще будет начать разработку заново, с нуля. И наконец, как быть, если программист всего один (он же схемотехник, конструктор, технолог, снабженец, отдел техподдержки, и менеджер по работе с клиентами в одном лице)? Скажете, так не бывает? Бывает.

Автор отнюдь не против найма высококвалифицированных специалистов. В конце концов, они действительно работают лучше других, у них есть чему поучиться менее опытным коллегам, они зачастую «тянут» на своих плечах большие и сложные проекты. Просто автор считает, что при планировании программного проекта нужно ориентироваться не на суперпрограммистов, а на средний уровень программистов в команде. И может быть, вместо того, чтобы давать суперпрограммисту маленькие части проекта, поручить ему руководство всем проектом? Пользы от этого явно будет больше.

## Время и деньги

Автор статьи [1] совершенно справедливо считает, что ключевое значение для успеха фирмы, занимающейся программированием, имеют планирование, определение стоимости работы и прогнозирование сроков ее исполнения. Однако все эти задачи в настоящее время далеки от разрешения. При достаточно большом объеме проблемы ни руководитель проекта, ни программист не сможет даже приблизительно назвать объем программы в тысячах строк. Они могут лишь догадываться, и то с определенной вероятностью, что программа будет иметь объем, скажем, больше тысячи сток и меньше пятнадцати тысяч. Сколько в ней окажется строк в конечном итоге, пять, шесть или восемь тысяч, можно будет сказать только после завершения проекта, задним числом. К тому же строки стокам рознь, можно написать тысячу строк очень простого кода за день, а можно провозиться три дня с небольшим, но сложным алгоритмом в пятьдесят строк. Между тем никакого объективного критерия трудоемкости разработки программного обеспечения, кроме количества строк программного кода, насколько известно автору, пока не придумано. А ведь сроки исполнения проекта, и в конечном итоге его стоимость, зависят именно от объема программы. Получается замкнутый круг: мы не можем определить трудоемкость (и, соответственно, стоимость) проекта до тех пор, пока он не будет закончен. Однако заказчик хочет знать стоимость работ заранее. Как быть? Можно воспользоваться верхней оценкой объема программы, но тогда сметная стоимость проекта и время исполнения станут слишком большими и заказчик уйдет к тем, кто пообещает ему более радужные перспективы. Можно воспользоваться средней оценкой, но тогда есть риск не вписаться в сроки и бюджет. К тому же любой руководитель проекта должен твердо помнить, что большинство программистов склонны завышать свою производительность, давая нижнюю оценку времени исполнения, поэтому названное программистом время лучше всего умножить на число «пи».

В свете вышеизложенного можно сформулировать некоторые предложения по повышению производительности труда коллектива разработчиков и снижению рисков фирмы.

## Демократия — это зло

Автору неоднократно приходилось наблюдать следующую ситуацию. Некая фирма проектирует и выпускает устройства на базе микроконтроллеров, имеющие сопряжение с персональным компьютером. Устройства непрерывно совершенствуются, получают новую функциональность, словом, работа кипит. Соответственно, в фирме есть программисты-микроконтроллерщики и программисты-компьютерщики, причем многие совмещают обе эти специальности. Более того, микроконтроллерщики занимаются и схемотехническими разработками, и сами выбирают микроконтроллер под свой проект. В результате каждый программист работает фактически на тех инструментальных средствах, на которых хочет работать. Один программирует на Delphi 7, другой никак не может «слезть» с Delphi 5, третий выбрал C++ Builder и пишет на нем. Один поставил в свое устройство PIC, другой без ума от Atmel AVR, третий является горячим поклонником чипов от Fujitsu. Проблем взаимодействия почти нет, ведь каждый специалист работает над своим проектом фактически в одиночку. Казалось бы, при такой организации процесса сплошные преимущества: разработчик использует те программные и аппаратные средства, которые ему хорошо известны, и, следовательно, его работа весьма эффективна, он практически не тратит времени на согласования своей части проекта с другими, поскольку работает над проектом самостоятельно, глубоко зная специфику своего проекта и его предметную область. В случае возникновения каких-либо проблем у заказчика разработчик всегда готов выехать в командировку и решить все вопросы. И разработчику хорошо — никто не «капает на мозги», все проектные решения он принимает сам, квалификация (а значит, и стоимость разработчика на рынке труда) растет непрерывно. Сложность приборов и программного обеспечения не очень велика, так что один специалист вполне справляется с проектом за полгода-год (или меньше, в зависимости от сложности проектов) без авралов, нервотрепки и сидений на работе до ночи. Тем более что всеми вопросами изготовления приборов, снабжения и прочими проблемами программисты, естественно, не занимаются, для этого фирма имеет штат технологов, конструкторов, снабженцев. В результате все счастливы, приборы и программы успешно разрабатываются, изготавливаются и отгружаются заказчику, фирма получает прибыль. Но вот уволился разработчик, и тут же выясняется, что заменить его трудно, а сроки сдачи проекта поджимают. Думаю, вы уже поняли причину. Программисты в этой фирме не взаимозаменяемы. Мало того что каждый программирует на чем хочет, они еще ведут проекты в одиночку, и все нюансы проекта находятся у разработчика в голове. «Дьявол кроется в деталях», говорят англичане. И эти детали нигде не зафиксированы. Единственное, что остается в фирме после ухода программиста, это ворох исходников да скудное описание проекта, сделанное в самых общих чертах, что называется, для галочки. И новый сотрудник, возможно, предпочтет переделать все заново.

## Одна фирма — один инструмент

Выводы просты. В пределах одной фирмы следует сократить ассортимент используемых инструментальных средств до абсолютно необходимого минимума. Должно использоваться всего два пакета: один для компьютерного программирования, второй для программирования микроконтроллеров. Соответственно, все проекты в фирме должны выполняться на одном семействе микроконтроллеров. Последнее требование выполнить не так уж и сложно. Как правило, фирма ведет разработку в какой-то одной области, поэтому можно выделить какие-то главные черты, которыми должен обладать микроконтроллер. Например, для бытовых устройств такой базовой характеристикой может служить цена, для измерительных приборов — наличие высокоточного встроенного АЦП, для промышленной автоматики — большое количество портов ввода-вывода и интерфейсов, для устройств с батарейным питанием — малая потребляемая мощность и наличие энергосберегающих режимов, для задач обработки сигналов — высокое быстродействие и наличие аппаратного умножителя. Просто выберите семейство и придерживайтесь его, благо в настоящее время большинство фирм выпускает в пределах одного семейства большой ассортимент микроконтроллеров с различным числом выводов, различными функциональными узлами и т. д.

Помимо «унификации» разработчиков, мы получаем дополнительную экономию средств, так как можем закупать однотипные комплектующие в бóльших количествах и, соответственно, по меньшей цене.

Если при приеме на работу новый сотрудник заявит, что он опытный программист, но ему не знакомы используемые вами инструментальные средства и микроконтроллер, просто предложите ему выбор: либо он их изучает во время испытательного срока, либо… ну, вы поняли. Для хорошего программиста «пересесть» с одного микроконтроллера на другой — вопрос максимум нескольких дней, с компьютерным программированием дело несколько сложнее, но на этом рынке рабочей силы и выбор больше.

## Программы нужно проектировать, а не писать

Хочется процитировать: «Как ведется проект? Написать код и отладить. Снова написать и снова отладить. И так до тех пор, пока проект не будет завершен» [1]. Представим, что нам нужно построить здание. Мы собрали бригаду строителей: каменщиков, бетонщиков, штукатуров, электриков, сантехников — словом, мастеров на все руки . И начали строить. Без проекта. Котлован выкопали на глазок, залили фундамент, не зная, сколько в здании будет этажей, планировку квартир выполнили, полагаясь на свой художественный вкус, причем половину здания построили из кирпича, половину — из железобетонных блоков. Электрик расположил лампочки и выключатели так, что в одних комнатах лампочек по пять штук, в других — ни одной, а все выключатели собраны в прихожей. Сантехник приварил трубы посреди гостиной, в метре от пола, при этом продолбив электропроводку. В результате здание рухнуло от хлопка дверью. Вы скажете, что так не делается. Любой знает, что здание должен спроектировать архитектор и инженеры-строители, каждый лист проекта должен быть согласован и утвержден, перед началом строительства составляют сметы и сетевые графики, а само строительство выполняется под строгим надзором специалистов и в точном соответствии с проектной документацией. И здания не падают (как правило). Программы же пишутся без проектной документации, и, соответственно, падают (зачастую). А если падают не сразу, то потому, что «здание» программы со всех сторон подперто бревнами и скреплено болтами в произвольных местах.

Отсюда вывод: программы нужно проектировать, а не писать [4, 5]. Причем проектировать со всей серьезностью, с оформлением проектной документации и утверждением ее руководителем проекта. Структура и алгоритмы программы, структуры данных и межмодульные интерфейсы должны быть описаны на бумаге до начала кодирования. Вот только одна проблема. Граф-схемы алгоритмов вызывают у большинства программистов стойкое отвращение, сформировавшееся еще на первом курсе института. И не зря. Квадратиками и ромбиками граф-схем просто невозможно описать более или менее сложную программу, рисовать их дольше, чем писать код, а сущности, относящиеся к структуре программы, вообще этим стандартом не предусмотрены. Но ведь есть UML! Великолепное средство документирования программных проектов, сочетающее разнообразные описательные средства и высокую степень гибкости. Литература по UML широко доступна [6–8], и здесь не имеет смысла подробно рассматривать этот графический язык программирования. Отметим только, что разработчику встроенного программного обеспечения следует обратить особое внимание на раздел диаграмм состояний UML. К тому же во многих случаях не стоит слишком строго и буквально придерживаться стандарта UML. Вы можете придумывать свои символы и обозначения, дополняя тем самым стандарт и приспосабливая его под собственные нужды.

Однажды изготовленная документация на ПО останется у вас навсегда, она понятна любому программисту, в ней отражены все аспекты структуры и поведения программы. Даже через несколько лет вы легко сможете модифицировать программу по ее графической документации. Просто представьте, что вам предстоит внести небольшое изменение в ассемблерный код длиной в несколько тысяч строк, написанный к тому же не вами. Без графической документации даже небольшая коррекция такого кода превращается в весьма нетривиальную задачу.

И дело не только в удобстве модификации программ. При наличии хорошо проработанной документации в виде диаграмм состояний написание программы превращается в хорошо формализованный, практически независящий от «человеческого фактора» процесс. И здесь очень полезной может оказаться SWITCH-технология (называемая также «программирование с явным выделением состояний», или «автоматное программирование»). Не останавливаясь подробно на данной технологии, можно сказать, что она основана на представлении программы в виде графа переходов конечного автомата. SWITCH-технология получила название от оператора switch языка C, который является основной структурой при написании программ в SWITCH-технологии.

SWITCH-технология разрабатывается в России с 1991 года. Ведущая роль в ее создании принадлежит А. А. Шалыто, профессору Санкт-Петербургского государственного университета информационных технологий, механики и оптики. Впервые SWITCH-технология была описана в книге А. А. Шалыто «SWITCH-технология. Алгоритмизация и программирование задач логического управления» [9] и получила дальнейшее развитие в работах [10–13]. Более подробно со SWITCH-технологией можно ознакомиться на сайте **http://is.ifmo.ru**.

## Программирование: искусство, ремесло или инженерия?

На вопрос, вынесенный в подзаголовок, каждый волен ответить по-своему. Для одного программирование — акт вдохновения, безудержный полет мысли. Для другого нет ничего важнее крепко сколоченного, добротно написанного кода, без претензий на гениальность. Третий любит и ценит инженерный подход, то есть структурированное, методичное решение поставленной задачи, подразумевающее глубокий ее анализ. А теперь внимание, правильный ответ. Программирование — это бизнес. Вам нужно получить возможно большую прибыль, продав программное обеспечение и «железо» как можно дороже, а потратить на его разработку как можно меньше (времени и денег). Поэтому в целях повышения эффективности производства программной продукции руководителю проекта (или фирмы) имеет смысл задуматься, какие программисты у него работают. При этом «искусствоведов» проще уволить сразу (вы же не хотите находиться в вечной зависимости от их приступов вдохновения), из «ремесленников» выйдут отличные кодеры (выше их пускать не стоит), «инженеры» могут стать аналитиками и проектировщиками (а вот кодировать у них получается не очень из-за привычки подолгу задумываться над каждой строкой, в результате чего код получается идеальный… по десять строк в день). Конечно, в чистом виде эти типы программистов встречаются редко, в каждом есть доля и того, и другого, и третьего, но какое-то качество все равно превалирует.

## Заключение

Отрасль программирования всегда была и остается весьма сложной. Ошибки в коде, срыв сроков, сильное влияние человеческого фактора — вот далеко не полный перечень вопросов, с которыми сталкивался, наверное, каждый программист. При производстве встроенного программного обеспечения значимость этих проблем возрастает на порядок. Повышенная ответственность разработчика перед заказчиком, сжатые сроки, недопустимость ошибок в коде делают «встроенное» программирование трудной задачей. Только превращение «интуитивного» программирования в четкий, отлаженный, «конвейерный» процесс позволит фирме выйти на новый уровень качества производства. Того самого, «статичного, бескризисного и всем понятного» производства, которое И. Кашенбойм поминает недобрым словом в [1]. Потому что лишь такое производство защитит вас от рисков, позволит планировать работу, избавит от зависимости от собственных сотрудников, повысит качество выпускаемой продукции, снизит трудоемкость и издержки.

Если данная статья вызовет интерес у читателей «Компонентов и технологий», автор мог бы в следующей статье описать процесс проектирования и документирования встроенного программного обеспечения на конкретном примере. ■

## Литература

1. Кашенбойм И. Квадрига Аполлона и микропроцессоры // Компоненты и технологии. 2006. № 4.
2. Константин Л. Человеческий фактор в программировании. СПб.: Символ-Плюс. 2004.
3. Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы. СПб.: Символ-Плюс. 2005.

4. Шалыто А. А. Новая инициатива в программировании. Движение за открытую проектную документацию // Мир компьютерной автоматизации. 2003. № 5.
5. Шалыто А. А. Еще раз об открытой проектной документации // PC Week/RE. 2005. № 11.
6. Боггс У., Боггс М. UML и Rational Rose. М.: Лори. 2001.
7. Фаулер М. UML. СПб.: Символ-Плюс. 2004.
8. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. М.: ДМК Пресс. СПб.: Питер. 2004.
9. Шалыто А. А. SWITCH — технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука. 1998.
10. Шалыто А. А. Логическое управление. Методы аппаратной и программной реализации. СПб.: Наука. 2000.
11. Шалыто А. А. Алгоритмизация и программирование задач логического управления. СПб.: СПбГУ ИТМО. 1998.
12. Шалыто А. А. Использование граф-схем и графов переходов при программной реализации алгоритмов логического управления //Автоматика и телемеханика. 1996. № 6, 7.
13. Гуров В., Нарвский А., Шалыто А. Исполняемый UML из России // PC Week/RE. 2005. № 26.