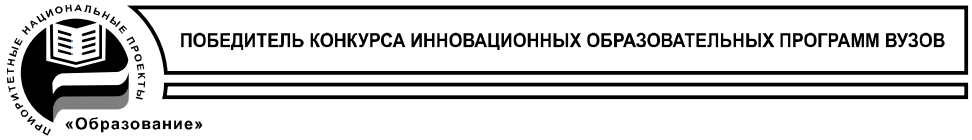
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ



А.О. Ключев, П.В. Кустарев, Д.Р. Ковязина, Е.В. Петров

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВСТРОЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Учебное пособие

Санкт-Петербург

2009

Ключев А.О., Кустарев П.В., Ковязина Д.Р., Петров Е.В. Программное обеспечение встроенных вычислительные систем. – СПб.: СПбГУ ИТМО, 2009. – 212 с.

Учебное пособие посвящено вопросам разработки программного обеспечения встроенных вычислительных систем (ПО ВВС). В пособии рассматриваются: организация системного и прикладного ПО ВВС, основные парадигмы и технологии программирования для ВВС, аппаратно-зависимое ПО ВВС, инструментальные средства проектирования ПО ВВС, отладка ПО ВВС, программирование ВВС с микроэнергопотреблением, технологии повторного использования при создании ПО ВВС.

Для подготовки бакалавров и магистров по направлению 23.01.00 «Информатика и вычислительная техника»; программы подготовки магистров 23.01.00.33 «Проектирование встроенных вычислительных систем» и 23.01.00.34 «Системотехника интегральных вычислителей. Системы на кристалле».

Рекомендовано к печати ученым советом факультета КТиУ, протокол №10 от 19.05.2009.

СПбГУ ИТМО стал победителем конкурса инновационных образовательных программ вузов России на 2007-2008 годы и успешно реализовал инновационную образовательную программу «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий», что позволило выйти на качественно новый уровень подготовки выпускников и удовлетворять возрастающий спрос на специалистов в информационной, оптической и других высокотехнологичных отраслях науки. Реализация этой программы создала основу формирования программы дальнейшего развития вуза до 2015 года, включая внедрение современной модели образования.

Оглавление

Введение

Информационно-управляющие и встраиваемые системы сегодня широко распространены. Управляющие компьютерные системы используются в бытовой технике (телевизорах, музыкальных центрах, холодильниках, стиральных машинах, микроволновых печах), в автомобилях, в современных средствах связи, в промышленности, в военной и аэрокосмической технике.

В начале восьмидесятых годов двадцатого столетия возникла тенденция к резкому увеличению капиталовложений в рынок встроенных систем. Это касается как промышленной и военной техники (традиционных экономических ниш подобного рода систем), так и постоянно расширяющегося рынка интеллектуальной бытовой техники. Большой спрос на управляющие системы вызвал серьезный прогресс в технологии производства полупроводников.

В отличие от систем общего назначения, проектирование различного рода управляющих систем накладывает на разработчиков дополнительную ответственность. В разработке данного программного обеспечения необходимо учитывать такие вещи, как надежность, безопасность, реальное время, ремонтопригодность, живучесть и так далее. В подавляющем большинстве случаев программное обеспечение встраиваемой системы нельзя рассматривать в отрыве от аппаратного обеспечения, конструкции и особенностей окружения, необходимо понимать, что проектируется не часть системы, а *система целиком*. Подходы, используемые современными программистами при создании больших программных систем общего назначения в мире встраиваемых систем, как правило, не работают или работают крайне плохо.

Материал учебного пособия излагается в общем, *концептуальном* плане, другими словами, речь пойдет об архитектуре встраиваемых систем и базовых принципах проектирования программного обеспечения. Главной целью лабораторных работ является осознание базовых механизмов, лежащих в основе системного, инструментального и прикладного программного обеспечения. Методом осознания является практическое прототипирование, т.е. создание простейших *действующих* встраиваемых систем.

Лабораторные работы в рамках данного курса выполняются не только на базе обычных офисных компьютеров, на которых будет базироваться инструментальная кросс система (компилятор, отладчик, симулятор и т.д.), но еще и на базе учебных микропроцессорных стендов (например, серии SDK). Практика показывает, что программное обеспечение, разработанное и отлаженное в идеальной среде с использованием симуляторов, на реальной аппаратуре, как правило, сразу не работает. Это создает определенные сложности в процессе преподавания, т.к. для отладки программного обеспечения студентов необходимо обеспечивать специализированным учебным оборудованием (осциллографами, микропроцессорными стендами,

прототипными платами и т.п.). Использование только персональных компьютеров не дает нужного эффекта в преподавании по целому ряду причин. К сожалению, современные операционные системы, применяемые для офисных приложений, не позволяют осуществлять работу в реальном масштабе времени. Большая часть механизмов, используемых во встраиваемых системах, либо отсутствует в персональном компьютере, либо фактически недоступна. Доступ к аппаратным средствам персонального компьютера закрыт ядром операционной системы, кроме того, необходимо отметить частую смену элементной базы ПК и ее большую сложность и избыточность.

В процессе обучения от студентов требуется понимание важнейших концепций, заложенных в программное обеспечение встроенных систем, минимальные умения в области их проектирования и практической реализации, а также осознание того, что встраиваемая система не может рассматриваться только как программная компонента.

В данном материале сравнительно мало практических советов, рецептов и примеров. Исключением является глава, описывающая знаменитый принцип «KISS». В ней дан ряд рецептов проектирования систем без каких-либо серьезных рассуждений о причинах возникновения тех или иных утверждений. Кроме того, некоторые рецепты приведены в главе, рассказывающей о концепции *стиля программирования*, предложенной профессором Непейвода. Все эти рецепты сформулированы людьми, стоящими у истоков современной вычислительной техники и проверены на практике. Тем не менее, не стоит воспринимать их как панацею от всех бед. Рецепты – лишь частный случай решения каких-то проблем. Студентам предлагается самостоятельно поразмышлять и доказать (или опровергнуть) приведенные утверждения.

В пособии практически не затрагиваются такие вопросы, как промышленные технологии проектирования, тестирования, отладки. Не рассматриваются также типовые варианты организации жизненного цикла ПО. Предполагается, что, имея фундаментальные знания, можно достаточно быстро изучить этот материал самостоятельно.

Описание конкретных программных и аппаратных систем сведено к минимально-возможному уровню. В учебном пособии большое внимание уделено таким понятиям, как *стиль программирования*, *модель вычислений*, *платформа* и *архитектура*. Чем это вызвано, почему не конкретные данные, а именно концепция?

Конкретные или частные знания очень быстро устаревают, всегда есть риск, что к концу обучения вы не увидите тех устройств или программных продуктов, которые проходили.

1. Большая часть идей в области вычислительной техники возникла довольно давно, несколько десятков лет назад. Сейчас происходит внедрение давно разработанных концепций. Практика показывает, что концептуальные знания стареют очень медленно.
2. Конкретной (частной) информации очень много, ориентироваться в ней

крайне сложно, если не знать, что именно нужно искать. Проблема в том, что человек – ограниченное в плане мыслительных способностей существо. Например, в один момент времени вы можете запомнить не больше десятка разнородных объектов, а в течении долгого времени сможете удержать в голове информацию не более чем 1000…5000 объектов. Т.е., уменьшая число объектов, с которыми работаете, вы освобождаете свой мозг для продуктивного мышления.

3. Зная концепцию и принцип работы базовых механизмов встраиваемой системы, понимая особенности аппаратного обеспечения и окружающей среды в комплексе, можно целенаправленно, коротким путем добраться до интересующей вас информации и, следовательно, быстро ее изучить.

Система реального времени

Принципиальное отличие обычных информационных систем от систем реального времени (СРВ) в трактовке реакции на входное воздействие. По отношению к СРВ говорят, что полученный поздно ответ приравнивается к неправильному ответу (по-английски «The right answer late is wrong»).

Приведем два определения термина «система реального времени».

- Система реального времени – вычислительная система с гарантированным временем реакции на события.
- Система реального времени – любая вычислительная система, в которой время формирования выходного воздействия является существенным.

Система реального времени не должна обязательно быть быстрой, это распространенное заблуждение. Система реального времени должна выдавать реакцию в ответ на информацию, поступающую на ее вход в гарантированные промежутки времени.

По степени важности последствий несоблюдения времени реакции выделяют две группы систем реального времени.

- Система мягкого реального времени.
- Система жесткого реального времени.

В первом случае несоблюдение требований реального времени не является катастрофическим по отношению к цели работы СРВ. Во втором случае, несоблюдение требований реального времени приводит к невозможности выполнения целевой функции системы.

Информационно-управляющая система

Информационно-управляющая система (ИУС) – цифровая система контроля или управления некоторым реальным объектом, называемым обычно объект управления. На вход ИУС поступает информация с датчиков, на выходе ИУС вырабатывается управляющее воздействие посредством исполнительного устройства.

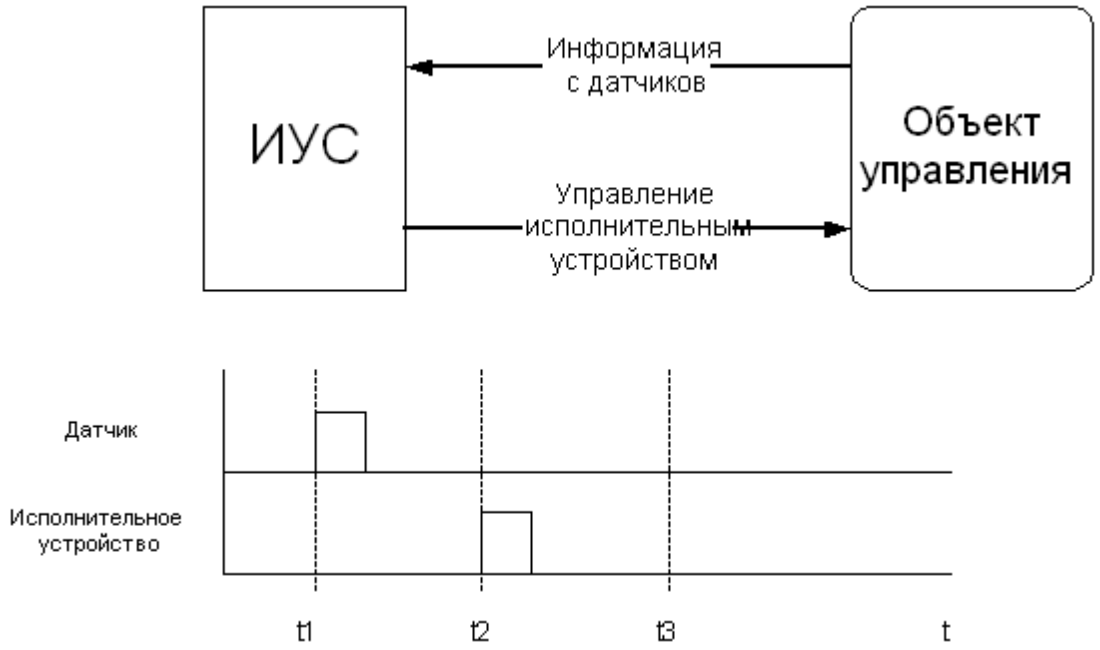


Рисунок 1. Взаимодействие ИУС и ОУ происходит в реальном масштабе времени

Информационно-управляющая система (или встроенная система) и система реального времени не являются синонимами. Отличие СРВ от ИУС состоит в том, что управляющая составляющая в СРВ не является обязательной. В класс СРВ может быть включена как информационно-управляющая система, так и система, относящаяся к информационным. Для примера к СРВ можно отнести компьютерные игры и системы резервирования авиабилетов.

Все ИУС являются СРВ, в которых реализованы мягкое или жесткое реальное время.

ИУС может быть реализована на базе одного или нескольких контроллеров. Таким образом, можно выделить два варианта построения ИУС.

- Сосредоточенные ИУС-ИУС, оборудование которых сконцентрировано в непосредственной близости от объекта управления и, как правило, конструктивно связано воедино.
- Распределенные информационно-управляющие системы (РИУС), ИУС-ИУС, разделенные на несколько устройств и расположенные вдали друг от друга, связанные посредством каких-либо каналов связи, например, контроллерной сети.

Сосредоточенные ИУС могут применяться тогда, когда система управления крайне простая или все вычислительные мощности можно сконцентрировать в одном месте. Достоинством сосредоточенных ИУС является высокая скорость взаимодействия между компонентами в тех случаях, когда вычислителей

(процессорных блоков) больше одного.

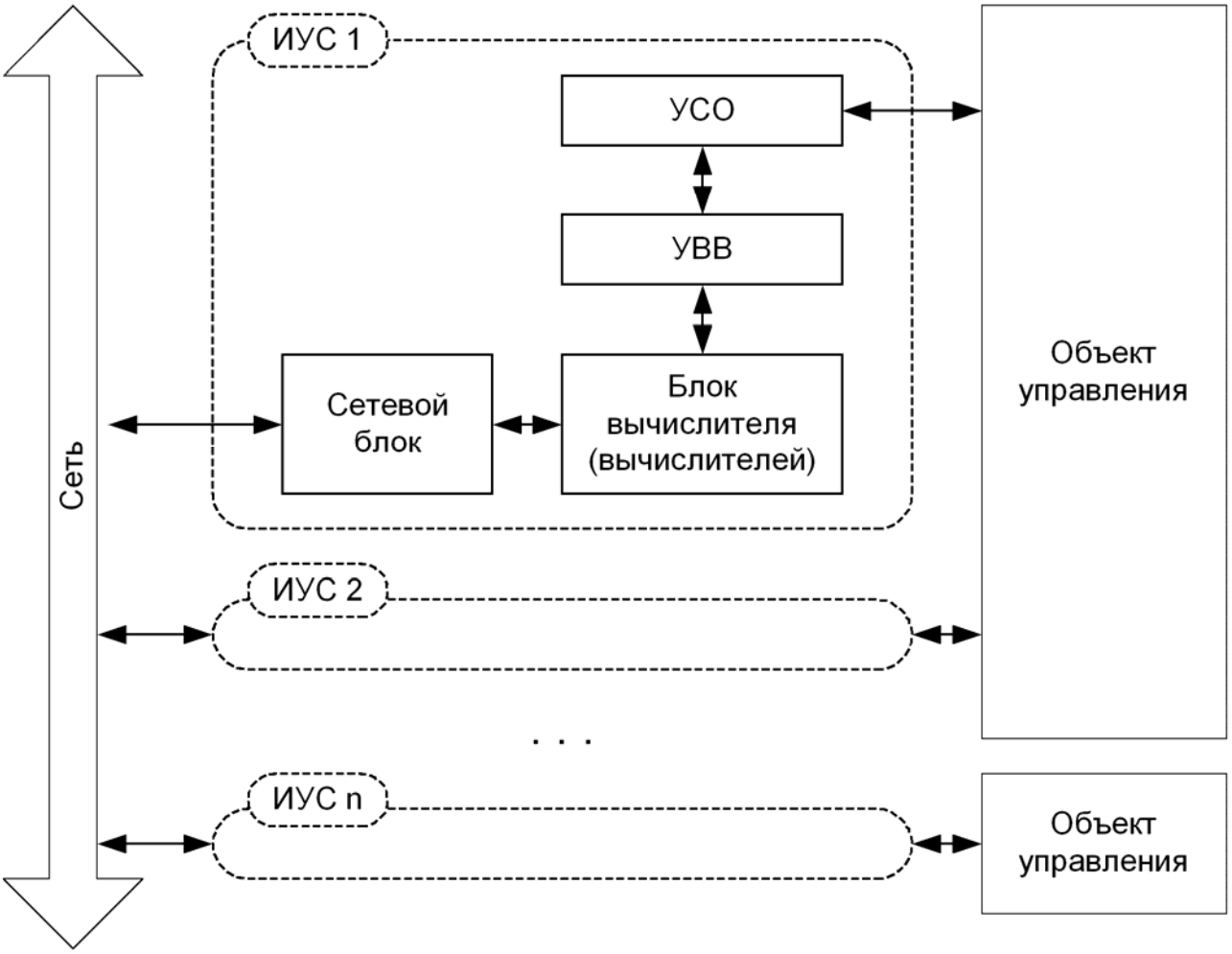


Рисунок 2. Распределенная ИУС состоит из нескольких ИУС, объединенных в единую сеть

Распределенные ИУС, как правило, применяются в тех случаях, когда прокладка кабелей от датчиков и исполнительных устройств обходится слишком дорого (или вообще невозможна) и проще поместить рядом с каждым объектом управления или его частью отдельный контроллер. Так как все контроллеры объединены в единую сеть, существует возможность централизованного управления такой системой. К сожалению, протяженные каналы связи, а также особенности реализации беспроводных каналов связи накладывают ограничения на скорость и качество передачи информации, что, в свою очередь, затрудняет соблюдение реального времени при передаче данных через сеть.

Необходимость работы в реальном масштабе времени (или просто в

реальном времени) является одной из главных особенностей работы ИУС. На графике показано три времени: t1 – время получения сигнала с датчика, t2 – время выдачи управляющего воздействия на исполнительный механизм, t3 – крайний срок выдачи управляющего воздействия. Если по какой-либо причине выдача управляющего сигнала задержится, сигнал будет выработан после t3, управляющий сигнал будет бесполезен или даже вреден. В качестве примера рассмотрим систему управления стеклоподъемником в автомобиле. Если ИУС игнорирует сигнал датчика положения стекла, то стекло, либо подающий его механизм могут быть испорчены.

В состав ИУС обычно входят следующие аппаратные и программные компоненты.

- Устройство ввода-вывода (УВВ).
- Устройство сопряжения с объектом (УСО).
- Блок вычислителя ИУС.
- Сетевой блок ИУС.
- Контроллерная сеть.
- Системное ПО, возможно ОС РВ.
- Инструментальное ПО.
- Прикладное ПО.

Встроенная система

В последнее время из-за прогресса в области вычислительной техники смысл термина *встроенная (или встраиваемая) система* видоизменился. Теперь это понятие стало близким по смыслу информационно-управляющей системе. Вариантов определений термина *встроенная система* много, но все они пока не отражают ее характерных особенностей.

Рассмотрим несколько вариантов определения.

- Встроенные вычислительные системы (ВВС) (embedded system) – специализированные (заказные) вычислительные системы (ВС), непосредственно взаимодействующие с объектом контроля или управления и объединенные с ним единой конструкцией.
- Встроенная вычислительная система (ВВС) (embedded system) – специализированная информационно-управляющая система (ИУС) для выполнения определенного набора функций.
- Встроенная вычислительная система (ВсС) – любая система, которая использует компьютер как элемент, но не выполняет функции компьютера. Примеры ВсС: DVD-проигрыватель, светофорный объект, банкомат, паркомат и т.д.
- Встраиваемой системой можно считать любую вычислительную систему, которая не является ПК, портативным компьютером или большим универсальным компьютером (mainframe computer).
- Встроенная вычислительная система – устройство, которое включает в себя программируемый компьютер, но не является при этом компьютером

общего назначения.

- Встроенная вычислительная система – любая вычислительная система, не являющаяся настольным компьютером.

Как правило, встроенная система является частью более крупной системы или встраивается непосредственно в объект управления.

Если в любой информационно-управляющей системе (ИУС) есть и информационный и управляющий аспекты, то во встроенной системе преобладает в основном управляющий аспект. Также как и ИУС, встроенная система является системой реального времени.

Встроенные системы – это системы «интегрированные» с объектами физического мира. Их элементы практически всегда ограничены по ресурсам. Это системы длительного жизненного цикла, часто автономные. Масштаб их по размерам и сложности меняется в очень широких пределах. Такие системы рассчитаны на непрофессиональных пользователей и часто выполняют критически важные функции.

По назначению встроенные системы можно подразделять на следующие категории.

- Системы автоматического управления (САУ).
- Измерительные системы и системы сбора информации с датчиков.
- Системы передачи данных (коммуникационные системы).
- Системы управления подвижными объектами.
- Подсистемы вычислительных систем общего назначения.

Примеры встроенных систем.

- Автопилоты морских судов и самолетов.
- Системы управления автомобилем (АБС, инжектор и т.п.).
- Сотовые телефоны.
- Медицинское оборудование.
- КПК.
- Игровые консоли.
- Цифровые музыкальные инструменты.
- Системы управления оружием.
- Сетевое оборудование (коммутаторы, маршрутизаторы, ADSL модемы и т.п.).

Диапазон реализаций ВсС очень велик. В него попадают и простейшие устройства уровня домашнего таймера, и сложнейшие распределенные иерархические системы, управляющие критически важными объектами. Важно, что, проектируя ВсС, разработчик всегда создает специализированную вычислительную систему, независимо от степени соотношения готовых и заново создаваемых решений. В сферу его анализа попадают все уровни организации системы. Он имеет дело не с созданием приложения в готовой операционной среде при наличии мощных и удобных инструментальных средств, а с созданием новой специализированной ВС в условиях жестких ограничений самого разного плана.

Безусловно, часть задач в области создания ВсС удается решать шаблонными способами, особенно, если речь идет о развитии или модификации уже готовой системы. Но даже в этом случае требуется использование качественной вычислительной платформы, мощного специализированного инструментария, тщательная верификация и тестирование продукта.

Задачи создания ВсС, которые не укладываются по тем или иным причинам в рамки шаблонных решений, постоянно требуют совершенствования методов и средств проектирования.

Распределенная встроенная система

Распределенная встроенная система – это пространственно рассредоточенная встроенная система. Такие системы характеризуются наличием слабой связи между компонентами.

Тенденция усложнения ВсС проявляется прежде всего в том, что большинство систем реализуются в виде многопроцессорных распределенных ВС или контроллерных сетей. Это дополнительно усложняет задачу проектировщика. Рассмотрим основные свойства современных распределенных ВсС.

- Множество взаимодействующих узлов (более двух). Интерес сегодня представляют системы с единицами тысяч взаимодействующих встроенных компьютеров.
- Работа в составе систем управления без участия человека. В таких системах оператор может присутствовать, получать информацию и частично иметь возможность воздействовать на работу системы, однако основной объем управления выполняет распределенная ВсС. Степень функциональной и пространственной децентрализации управления может меняться в широких пределах. Вычислительные элементы ВсС выполняют задачи, отличные от задач вычислений и коммуникаций общего назначения.
- Распределенные ВсС используются в составе больших по масштабу технических объектов (например, самолет или здание) или взаимодействуют с объектами естественной природы (например, комплексы мониторинга окружающей среды).
- Распределенные ВсС могут характеризоваться узлами с ограниченным энергопотреблением, иметь фиксированную или гибкую топологию, выполнять критичные для жизнедеятельности человека функции, требовать высокотехнологичной реализации или создаваться как прототип.

1. Организация системного и прикладного программного обеспечения ВВС

1.1. Специфика ВВС

Особенности организации и использования ВВС налагают определенные ограничения на организацию программного обеспечения. К таким особенностям можно отнести следующие:

- небольшие аппаратные ресурсы;
- реальный масштаб времени;
- соблюдение повышенных требований по надежности и безопасности.

1.2. Влияние особенностей аппаратного обеспечения ВВС на организацию ПО

Особенности контроллеров

Основными наиболее сложными и ответственными компонентами, из которых строят встраиваемые системы, являются контроллеры. Как правило, контроллер – некое законченное устройство. Фактически, это специализированный, ориентированный на управление компьютер. Контроллер может управлять функционированием блоков и внешних устройств компьютера общего назначения (например, контроллер НЖМД), двигателем автомобиля, микроволновой печью, прокатным станом или радиотелескопом.

Контроллеры могут относиться к классу информационно-управляющих систем (ИУС), встроенных систем, систем на кристалле (СнК), IP компонент или интегральных микросхем. В распределенных информационно-управляющих системах контроллеры объединяются в контроллерные сети. К промышленным и бортовым контроллерам предъявляются повышенные требования по надежности и безопасности. Так как контроллеры в основном занимаются управлением реального объекта (объектом управления), к ним выдвигается требование соблюдения реального времени.

Приведем классификацию контроллеров. По месту использования различают следующие виды контроллеров:

- промышленный контроллер;
- приборный контроллер;
- бортовой контроллер;
- контроллер в интегральном исполнении;
- микроконтроллер.

По степени программируемости различают:

- программируемый логический контроллер (ПЛК);
- программируемый контроллер;
- специализированный контроллер.

Рисунок 3. Промышленный контроллер uPAC-7186EG-G

Рисунок 4. Промышленный контроллер i-8411

Рисунок 5. Промышленный контроллер ADAM-5000

К специфическим особенностям контроллеров можно отнести следующие.

- Сравнительно малые аппаратные ресурсы (ОЗУ, память программ, быстродействие процессора). Этот факт объясняется повышенными требованиями к энергопотреблению и надежности. В контроллерах стараются не использовать динамическую память и процессоры с высокими тактовыми частотами. По причине низкой надежности, для хранения данных стараются не использовать накопители на жестких дисках.
- Отсутствие возможности непосредственного программирования из-за небольших аппаратных ресурсов (физически негде развернуть компилятор отладчик и и редактор) вызывает необходимость в дополнительной инструментальной машине. Этот факт усложняет процесс отладки и тестирования ПО и требует более высокой квалификации программистов.
- Отсутствие (в большинстве случаев) интерфейса пользователя затрудняет процесс отладки. В качестве интерфейса пользователя в контроллерах используется несколько светодиодных индикаторов, семисегментные индикаторы или небольшие ЖКИ.
- Специфика использования контроллеров (например, в ряде ответственных применений) налагает особые требования на программное обеспечение. В частности, предъявляются повышенные требования по надежности, безопасности и соблюдению реального масштаба времени. Как правило, обычные для информационных систем способы проектирования ПО для контроллеров не подходят.

Микроконтроллер

Микроконтроллер или однокристальная микро-ЭВМ (ОКМЭВМ) – контроллер в интегральном исполнении, реализованный на микропроцессорной

элементной базе. Микроконтроллер в одном кристалле содержит микропроцессор и набор периферийных устройств и контроллеров: контроллер прерываний, таймеры, контроллер сети, контроллер последовательного канала, контроллер памяти, контроллер ПДП и так далее.

Система на кристалле

Системы на кристалле (СнК, System on a Chip, SoC) – в общем случае системы, на едином кристалле которых интегрированы процессор (процессоры, в том числе специализированные), некоторый объем памяти, ряд периферийных устройств и интерфейсов, – то есть максимум того, что необходимо для решения задач, поставленных перед системой. Выражение "система на кристалле" не является, строго говоря, термином. Это понятие отражает общую тенденцию к повышению уровня интеграции за счет интеграции функций.

Производительность приборов класса "система-на-кристалле" в значительной мере зависит от эффективности взаимодействия всех встроенных компонентов и от эффективности их взаимодействия с внешним, относительно прибора, миром. В первую очередь, это связано с различием в быстродействии встроенных компонентов, в особенности организации интерфейсов.

Системы на кристалле обычно состоят из трех основных цифровых системных блоков: процессор, память и логика. Процессорное ядро реализует поток управления, когда каждой управляющей программой однозначно устанавливаются последовательности выполнения операций обработки данных, что позволяет задавать один из возможных алгоритмов работы всей интегральной схемы. Память используется по ее прямому назначению – хранение кода программы процессорного ядра и данных. Наконец, логика используется для реализации специализированных аппаратных устройств обработки и прохождения данных, состав и назначение которых определяются конечным приложением – потоком данных.

Можно считать, что современные микроконтроллеры являются примерами СнК, производимыми крупными сериями.

Классификация микроконтроллеров

Существует множество способов, с помощью которых можно производить классификацию микроконтроллеров.

- По разрядности различают 8, 16 и 32 разрядные микроконтроллеры.
- По возможностям в области обработки сигналов можно рассматривать обычные микроконтроллеры и DSP-микроконтроллеры.
- По области применения различают следующие микроконтроллеры: автомобильные, промышленные, для контроллерных сетей, управления двигателями, управления беспроводными сетями.

По объему вычислительных ресурсов условно можно выделить четыре характерные группы микроконтроллеров.

- Периферийные процессоры – Microchip PIC 10, PIC12, PIC16, PIC18,

PIC24, Atmel AT90xxxx и т.п.
- Универсальные 8-ми и 16-ти разрядные ОМЭВМ – Intel MCS51, Siemens SAB 5xx, Atmel Mega10x и т.п.
- Универсальные 16-ти и 32 разрядные ОМЭВМ – Fujitsu FR-50, ARM7 и т.п.
- Универсальные однокристальные 32-х разрядные микроконтроллеры и процессоры Freescale MPC560xx, ARM9, ARM11 и т.п.

Для первой категории процессоров характерны следующие особенности:
- небольшой объем памяти данных (десятки – сотни байт);
- небольшой объем памяти программ (единицы – десятки килослов);
- сравнительно высокое быстродействие;
- система команд RISC;
- низкое энергопотребление;
- малое число выводов;
- невозможность подключения внешней памяти;

Старшие модели микроконтроллеров могут иметь в своем составе сетевые контроллеры. Основная идея в этих контроллерах – обеспечение создания устройств с низким энергопотреблением и минимальным количеством компонентов на плате.

Для второй категории процессоров характерна возможность использования внешней (внекристальной) памяти. Отличает эту категорию низкая цена и небольшие вычислительные ресурсы. Производительность таких микроконтроллеров как правило значительно ниже, чем у первой категории. Контроллеры этого типа применяются в основном в простых и дешевых устройств, не предъявляющих повышенных требований по производительности и энергопотреблению, но имеющих повышенные требования по объему программного кода и требуемой памяти данных.

Третья категория процессоров имеет развитые аппаратные средства для повышения производительности обработки информации, гораздо более мощный, по сравнению с первой и второй категориями центральный вычислитель и расширенное адресное пространство. Начиная с этой категории в состав микроконтроллеров производители начинают наиболее активно включать сетевые контроллеры. В настоящее время это наиболее распространенные микроконтроллеры.

В четвертой категории процессоров характерно применение механизмов защиты памяти и большое адресное пространство, что позволяет без особых проблем применять операционные системы реального времени. От третьей категории их также отличает более высокая производительность.

1.3. Анализ блоков микроконтроллеров с точки зрения программирования

Ниже перечислен ряд характерных блоков, входящих в большинство микроконтроллеров и СнК. Мы рассмотрим эти блоки в общем виде. Более

подробную информацию о конкретной реализации тех или иных механизмов можно узнать в руководствах пользователя (user manual), выпущенных производителями для своей продукции.

Центральный процессор

Центральный процессор является одним из главнейших элементов управляющей системы. Выбор процессора сильно влияет на архитектуру как аппаратной, так и программной части. Какие особенности имеются у процессоров, применяемых в ИУС?

4. В ИУС, как правило, используются специализированные процессоры для встроенных применений. В частности, в небольших (или простых и дешевых) системах часто используют однокристальные микроЭВМ, с интегрированной в один кристалл памятью и периферийными устройствами.
5. Центральный процессор встраиваемых систем ориентирован на управление. Большая часть процессоров для встраиваемых применений серьезно отстает от процессоров общего назначения в области обработки больших массивов данных, но зато значительно превосходит их при обработке аналоговых и цифровых сигналов.
6. Ядро процессора для встраиваемых применений должно быть построено на основе статических схем, в которых запоминающие ячейки сделаны на базе триггера. Такое решение дороже динамических (на базе конденсаторов, заряд в которых нужно постоянно регенерировать), но позволяет гибко менять тактовую частоту, что в свою очередь дает возможность управлять энергопотреблением.
7. Связь с периферийными устройствами производится в основном через регистры специального назначения, так называемые SFR (Special Function Register). В зависимости от типа микроконтроллера, эти регистры могут находиться в адресном пространстве регистров общего назначения или в общем адресном пространстве. От места расположения регистра зависит размер команды (чем больше регистров, тем длиннее должно быть поле адреса в команде). Как правило, если регистров много, они располагаются в общем адресном пространстве.
8. Процессоры для встроенных применений выпускаются для расширенного температурного диапазона.
9. Для надежного функционирования ОС РВ, как правило, необходим режим защиты памяти, т.е. поддержка изолированных адресных пространств для каждого процесса. Этот механизм присутствует в наиболее дорогих и сложных процессорных ядрах, например, ARM9.
10. Для эффективного переключения контекстов необходима соответствующая поддержка в процессоре, такая как банки регистров, специальные

команды для сохранения фреймов стека и т.п.

11. Для своевременной реакции на различные события в процессоре для встраиваемых применений существует развитая система прерываний. Время входа в прерывание стараются сделать минимальным, для этого используют специальные решения, связанные, например, с банками регистров. При использовании нескольких регистровых банков, экономия времени может достигаться тем, что во время вызова обработчика прерывания не происходит сохранения в стек контекста прерванной задачи, так как у каждого прерывания есть свой собственный регистр банков. При использовании операционных систем реального времени такое решение теряет смысл, так как на большое количество процессов регистровых банков скорее всего не хватит.

Система контроля питания

Система питания является фундаментом для любой электронной схемы. К сожалению, во встроенных системах не всегда удается добиться качественного вторичного электропитания при использовании автономных источников энергии (аккумуляторов, химических элементов питания и т.п.), бортовой сети (например, в автомобиле), при наличии большого количества помех (например, на производстве). Система контроля питания предназначена для обеспечения надежного функционирования микроконтроллера в условиях нестабильного питающего напряжения.

Сразу после включения питания устройства на плате начинаются переходные процессы. Нарастание напряжения происходит не мгновенно и не линейно, время установки стабильного напряжения питания зависит от схемы и составляет обычно десятки-сотни миллисекунд. На этот момент времени система контроля питания задерживает старт микроконтроллера. Если старт не задержать, микроконтроллер, получая нестабильное питание, может давать сбои в работе и часто рестартовать.

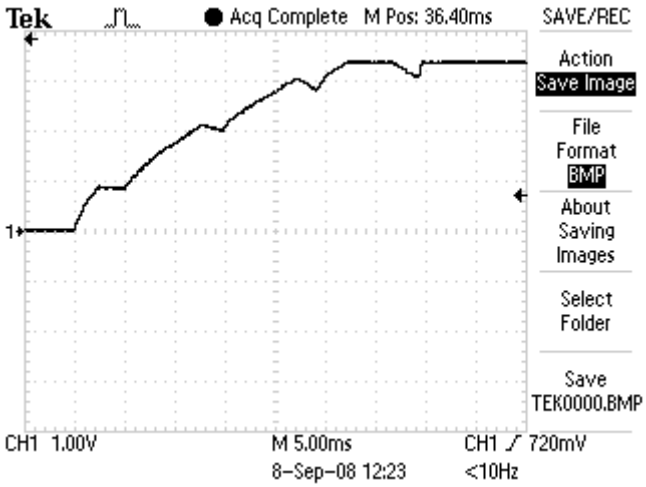


Рисунок 6. Осциллограмма, снятая с шины вторичного питания контроллера во время включения

В процессе работы система контроля питания постоянно проверяет уровень напряжения в цепях питания. Если уровень отклоняется от заданной величины, система контроля питания вырабатывает прерывание. Обработчик прерывания может корректно завершить работу встраиваемой системы, например, при внезапном пропадании питания.

Что можно сделать в обработчике прерывания, когда произошел сбой питания? Можно попытаться сохранить в энергонезависимой или обычной памяти контроллера текущее состояние прикладной программы, чтобы после возобновления подачи электроэнергии продолжить работу с прерванного места. Естественно, для реализации такого механизма защиты от сбоев питания вы должны реализовать свою программу так, чтобы в ней были четко выражены состояния ее работы. Другими словами, при проектировании такого рода программ очень полезно использовать конечные автоматы.

Подключение внешней памяти

Современный уровень развития технологий позволяет встраивать в микроконтроллеры достаточно большие объемы оперативной и FLASH памяти. К сожалению, при проектировании встроенных систем приходится минимизировать количество микросхем на плате. Дело в том, что каждая микросхема – это дополнительный потребитель энергии. Кроме того, увеличение количества элементов приводит к повышению вероятности выхода устройства из строя, т.к. каждый новый элемент – потенциальный источник ошибок. Представьте, что каждый вывод любой радиодетали может потерять контакт с печатным проводником платы под воздействием вибрации или какой-либо агрессивной среды. Чем больше контактов, тем больше вероятность сбоя всего устройства. Подключение внешней памяти серьезно усложняет разводку печатной платы, что может привести к ошибкам трассировки или необходимости выпуска более дорогой, многослойной платы. Тем не менее, в ряде случаев не обойтись без внешнего ОЗУ и FLASH из-за большого объема

требуемых ресурсов.

В большинстве современных микроконтроллеров есть возможность подключения внешней памяти. Для этого наружу выведены шина данных, адреса и управления. Кроме микросхем ОЗУ и FLASH к шине внешней памяти можно подключать также и различные устройства, например, контроллер Ethernet или видеоконтроллер (см. схему электрическую принципиальную учебного стенда SDK-1.1).

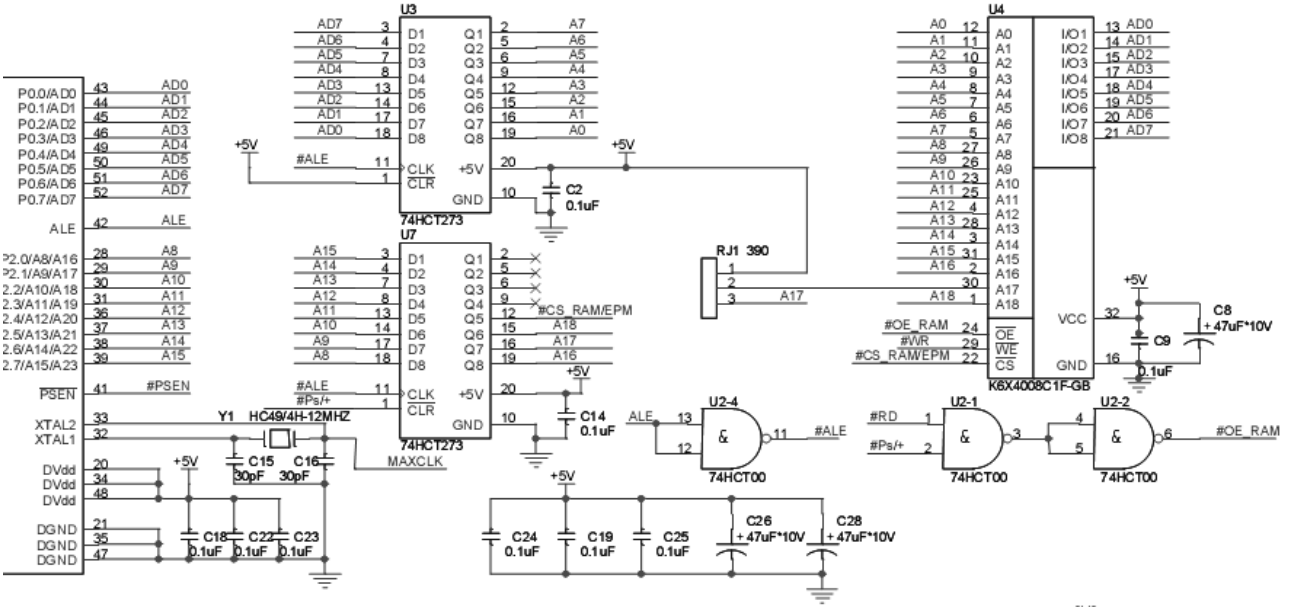


Рисунок 7: Подключение внешнего ОЗУ к микроконтроллеру ADuC 812 в стенде SDK-1.1

При обращении к определенной области адресного пространства должен вырабатываться сигнал выбор кристалла (Chip Select, CS). Выдачу такого сигнала обеспечивает программируемый блок адресной селекции.

Блок адресной селекции есть в сравнительно мощных микроконтроллерах, таких как Philips LPC2000 (см. схему электрическую принципиальную SDK-2.0). В ADuC 812 он отсутствует. Если бы в SDK-1.1 больше устройств было подключено к шине, потребовался бы внешний адресный селектор.

Микросхемы памяти, а также различные контроллеры, подключаемые как память (в виде набора регистров управления), имеют различные характеристики по быстродействию. Информацию о возможностях той или иной микросхемы можно узнать в спецификации, предоставляемой производителем микросхем (Data-sheet). Как правило, контроллер памяти позволяет в широких пределах устанавливать такты неготовности, ширину шины данных, управлять возможностью записи (имеется в виду запрещать запись) для сопряжения с устройствами, обладающими разными характеристиками (ОЗУ, FLASH, ПЗУ, контроллер Ethernet и т.п.).

Особенности тактового генератора, делитель частоты

Схемотехника, лежащая в основе современных микроконтроллеров, позволяет им работать в широком диапазоне частот, вырабатываемых тактовым генератором. Для обеспечения гибкого управления тактовой частотой используются делители и умножители частоты.

Для управления тактовой частоты используют фазовую автоподстройку частоты (ФАПЧ, Phase-locked loop, PLL).

Фазовая автоподстройка частоты (**ФАПЧ**) – система автоматического регулирования, подстраивающая частоту управляемого генератора так, чтобы она была равна частоте опорного сигнала. Регулировка осуществляется благодаря наличию отрицательной обратной связи. Выходной сигнал управляемого генератора сравнивается на фазовом детекторе с опорным сигналом, результат сравнения используется для подстройки управляемого генератора [23].

Блоки управления тактовой частотой являются конфигурируемыми и позволяют управлять не только тактовой частотой центрального процессора, но и раздельно регулировать тактовые частоты внутренних периферийных устройств и внешней шины.

В приложении 1 приведен пример программы для настройки PLL микроконтроллера из сеймейства FR50 фирмы Fujitsu.

Средства понижения энергопотребления

Практически все современные микроконтроллеры имеют встроенные средства понижения энергопотребления, позволяющие отключать не используемые в данный момент блоки, понижать тактовую частоту процессора и переходить в различные режимы сна. Подробнее об этих средствах можно прочитать в соответствующем разделе данной книги.

Часы реального времени

Во многих, сравнительно мощных микроконтроллерах есть встроенный блок часов реального времени (Real-Time Clock, RTC). Часы позволяют отсчитывать дату и время. Как правило, блок RTC создают на элементной базе, обеспечивающей пониженное энергопотребление. У микроконтроллера обычно существует возможность подключения дополнительного электропитания (например, литиевой батарейки или ионистора). Пример микроконтроллера со встроенным RTC – Philips LPC 2000, на базе ядра ARM7. На этом контроллере построен учебный стенд SDK-2.0. Подробнее об особенностях этих часов можно посмотреть в главе «Real-Time Clock» руководства пользователя микроконтроллеров LPC2119/2129/2194/2292/2294.

Встроенная FLASH память

Встроенная FLASH память в основном предназначена для хранения программ и сравнительно больших объемов данных. Объем памяти, в зависимости от типа микроконтроллера, может достигать от единиц до сотен килобайт.

FLASH память не позволяет перезаписывать отдельные байты. Адресное пространство памяти разбито на участки по аналогии с жесткими дисками, называемыми секторами. Для записи новой информации необходимо с помощью специальной команды стереть целый сектор или все сектора сразу. Один из секторов называется загрузочным. Обычно он располагается в той части адресного пространства микроконтроллера, с которого стартует процессор. Адрес старта – очень важный параметр, его обязательно нужно знать для всех используемых микроконтроллеров.

Необходимо помнить, что количество циклов записи и стирания ограничено и достигает десятков тысяч – миллионов раз, и если вы собираетесь использовать FLASH память для хранения данных необходимо предусмотреть.

Для обеспечения операции стирания и записи используются команды, отправляемые по специальным адресам адресного пространства FLASH памяти.

В приложении 2 приведен пример программы для записи внутренней FLASH памяти для микроконтроллеров PIC 16 фирмы Microchip.

Встроенная энергонезависимая конфигурационная память

Встроенная энергонезависимая память (non-volatile memory) создается по разным технологиям. EEPROM обычно имеет ограничение на количество циклов записи в десятки тысяч – миллионы раз. Если энергонезависимая память является сегнетоэлектрической (FRAM), то количество циклов записи может быть гораздо большим (миллиарды циклов записи).

Встроенная энергонезависимая память предназначена для хранения сравнительно небольшого (десятки – сотни байт) количества информации (например, конфигурационной). Этот блок обычно присутствует в маломощных микроконтроллерах для уменьшения необходимого количества микросхем и соединений на плате, с целью удешевления и снижения энергопотребления. Доступ к ячейкам памяти байтовый. Для чтения и записи обычно используются регистры специального назначения.

Сторожевой таймер

Допустим, в нашей системе есть некий процесс, выполняющий какую-либо полезную задачу. Назовем его прикладным или наблюдаемым процессом. Если процесс выполняет важную целевую функцию, то прекращение его деятельности может привести к остановке работы всей встроенной системы, то есть к сбою. Если не принять особых мер по обнаружению таких ситуаций, встроенная система может работать некорректно в течении достаточно долгого времени. Для вывода систем из сбойного состояния и приведения её в нормальный режим функционирования обычно используют сторожевой таймер.

Таким образом, сторожевой таймер является механизмом защиты системы от сбоев.

Рассмотрим работу сторожевого таймера подробнее. В работе участвуют три процесса: наблюдаемый прикладной процесс, процесс сторожевого таймера и служебный процесс, реализующий механизм защиты системы от аварийных ситуаций. Суть механизма сторожевого таймера состоит в проверке критерия, по которому можно определить, что наблюдаемый процесс работает нормально. Если сторожевой таймер обнаруживает, что с наблюдаемым процессом все в порядке, то ничего не происходит. Если сторожевой таймер определил, что с наблюдаемым процессом что-то не так, происходит передача информации системе разрешения аварийных ситуаций, которая, в свою очередь, принимает решение о дальнейшей судьбе наблюдаемого процесса.

В самом примитивном варианте в качестве сторожевого таймера выступает обычный вычитающий счетчик. Во время инициализации в счетчик записывается какое-либо значение. Если в процессе работы в счетчик эпизодически вносится новая константа, то ничего не происходит. Если же прикладной процесс не успевает записать константу, и счетчик успевает досчитать до нуля, вырабатывается сигнал аппаратного рестарта и процессор перезапускается. Естественно, простой вариант реализации сторожевого таймера не является 100% гарантией от выхода системы из сбойного состояния.

Рассмотрим типичную ошибку, допускаемую программистами при использовании простого сторожевого таймера. Для того, чтобы контролируемый процесс прекратил работу, достаточно вставить код обновления счетчика сторожевого таймера внутрь цикла проверки, который будет всегда давать ложное значение из-за какой-либо ошибки (в аппаратуре или программе).

```
while( device_ready() == 0      ) reset_watchdog_timer();
```

Если функция devive_ready всегда будет возвращать ноль, то мы бесконечное время будем сбрасывать сторожевой таймер и система будет всегда

находится в состоянии сбоя.

Нужно заметить, что в большинстве микроконтроллеров реализована самая простая схема работы сторожевого таймера. Для её нормальной работы нужна дополнительная, сложная программная обработка.

Ещё одним неудобством простой схемы сторожевого таймера является выработка аппаратного сигнала рестарта (RESET). В большинстве случаев гораздо корректнее производить повторную инициализацию или рестарт одной или нескольких взаимосвязанных частей системы, а полный горячий рестарт осуществлять только при каких-либо фатальных сбоях.

Система прерываний

Прерывание – это прекращение выполнения текущей команды или последовательности команд для обработки некоторого события обработчиком прерывания, с последующим возвратом к выполнению прерванной программы. Управляет прерываниями специальный контроллер – контроллер прерываний. Такой контроллер есть практически во всех микроконтроллерах, даже в самых простых. Управление контроллером прерываний осуществляется через регистры. Каждому прерыванию можно задать приоритет, численно определяющий важность события.

Каждому прерыванию ставится в соответствие обработчик – специальная программа, предназначенная для обработки прерывания. Как правило, адреса обработчиков располагаются в специальной таблице, так называемой таблице векторов прерываний.

Ниже приведен исходный текст обработчика прерывания UART для микроконтроллера MB90F590 фирмы Fujitsu.

```
/*-------------------------------------------------------------
Обработчик прерываний по приему байта по последовательному каналу.
------------------------------------------------------------- */
__interrupt void Serial0RxHandler(void)
{
    BYTE tmp;
    tmp = USR0;
    if ( tmp&0x60 )
        UMC0_RFC = 0;
    if ( tmp&0x80 )
    {
        tmp = UIDR0;

        if ( rx_count < RX_FIFO_SIZE )
        {
            rx_buf[(rx_start+rx_count)&(RX_FIFO_SIZE-1)] = tmp;
            ++rx_count;
        }
    }

}
```

Контроллер прямого доступа к памяти

Контроллер прямого доступа к памяти (ПДП, Direct Memory Access, DMA) – блок, позволяющий разгрузить центральный процессор во время пересылок между устройством ввода-вывода и памятью. Обычно блоками ПДП снабжаются сравнительно мощные модели микроконтроллеров. В основном ПДП используется для взаимодействия памяти с устройствами ввода-вывода, которые могут создать большой поток данных: сетевыми контроллерами, UART, ЦАП и АЦП.

Как правило, блок ПДП программисты встроенных систем, особенно начинающие, стараются обходить стороной, считая его излишне сложным. На самом деле, использование ПДП не является более сложным, чем, например, использование системы прерываний, а эффект от применения ПДП может разгрузить центральный процессор и увеличить производительность контроллера.

Таймер

Таймер позволяет производить отсчет временных интервалов заданной продолжительности. Принцип действия таймера основан на двоичном счетчике, с возможностью предварительной записи исходного значения. После каждого такта синхросигнала счетчик прибавляет или отнимает единицу от имеющегося у него значения. При достижении нуля (то есть при переполнении), счетчик вырабатывает активный уровень на выходе. Как правило, выходной сигнал таймера заводят на вход запроса прерывания микропроцессора или контроллера прерываний.

В большинстве современных встроенных систем таймеры используются в

качестве основы для организации системы разделения времени на базе переключателя задач. В данном случае таймер используется в паре с механизмом прерываний.

Устройство захвата-сравнения

Устройство захвата-сравнения предназначено для измерения периодов и длительностей импульсов, скважности, а также для генерации импульсов. С помощью этого модуля можно получить широтно-импульсную (ШИМ) и фазовую модуляцию (ФМ).

Порт ввода-вывода

Порт ввода-вывода – это точка подключения микроконтроллера к внешним устройствам ввода-вывода. Порты бывают дискретные и аналоговые.

Дискретный порт ввода-вывода позволяет формировать и принимать сигналы, которые кодируются логическими значениями «0» и «1». Логическому нулю соответствует напряжение, близкое к нулю, а логической единице – напряжение, близкое к напряжению питания.

Напряжение, примерно соответствующее средней части диапазона «питание – корпус», при считывании дает непредсказуемый результат.

Каждый дискретный порт можно запрограммировать на вход или на выход. Кроме этого, во многих микроконтроллерах можно указать, будет ли использоваться порт по схеме с открытым или закрытым коллектором.

Аналоговый порт ввода-вывода предназначен для работы с аналоговым (непрерывным) сигналом. В отличие от дискретного сигнала, принимающего всего два значения, напряжение аналогового сигнала может иметь любое значение (в определенных пределах) и меняться во времени. Пример аналогового сигнала – синусоида. Такую форму, например, имеет электрическое напряжение в сети 220 В 50 Гц.

Для оцифровки аналогового сигнала используют аналого-цифровой преобразователь (АЦП). Для обратного преобразования цифрового кода в аналоговый сигнал – цифро-аналоговый преобразователь (ЦАП). Часто в микроконтроллерах вместо ЦАП используют модули захвата-сравнения, с помощью которых реализуют широтно-импульсный модулятор (ШИМ). Аналоговый сигнал получается на выходе после обработки сигнала, полученного с выхода ШИМ с помощью интегрирующей схемы.

В большинстве микроконтроллеров каждый вывод может выполнять две или три различные функции, например, дискретный порт ввода-вывода, аналоговый выход и последовательный канал. С помощью специального конфигурационного регистра можно выбрать нужную функцию. С точки зрения безопасности аппаратуры, лучше всего производить настройку портов ввода-

вывода сразу после старта микроконтроллера.

Контроллер последовательного интерфейса

Контроллер последовательного интерфейса (UART) присутствует практически во всех микроконтроллерах, за исключением совсем маленьких. UART позволяет связать микроконтроллер с компьютером через физический интерфейс RS232, для того, чтобы подключить терминал, программатор, отладчик или программу для сбора данных с контроллера, организовать контроллерную сеть с помощью физического интерфейса RS-485.

Контроллер последовательного канала преобразует байт, переданный параллельным кодом в последовательность нулей и единиц. Точно так-же, последовательность битов, полученная UART, преобразуется в байты параллельного кода. Для передачи и приема данных по последовательному каналу в минимальной конфигурации достаточно всего трех проводов: передача (Tx), прием (Rx) и общий провод (GND).

Ниже приведены примеры программ для записи (wsio) и чтения (rsio) байтов. Программа написана для учебного стенда SDK-1.1, сделанного на базе микроконтроллера AduC 812 (ядро Intel MCS51).

```
void wsio( unsigned char c )
{
    SBUF = c;
    TI   = 0;
    while( !TI );
}
unsigned char rsio(void)
{
    while( !RI );
    RI = 0;
    return SBUF;
}
```

Контроллер интерфейсов I2C и SPI

Контроллер интерфейсов I2C и SPI позволяет подключать различные устройства, такие как микросхемы памяти (EEPROM), часы реального времени, АЦП и т.п. к микроконтроллеру через минимальное количество проводов. Интерфейсы SPI, в особенности I2C, значительно проигрывают параллельной шине в скорости передачи данных, зато позволяют сэкономить на разводке печатной платы. Обычно эти интерфейсы применяются в тех случаях, когда высокая скорость передачи не нужна, а устройство должно быть компактным и недорогим.

Контроллер CAN

Контроллер CAN включается в состав большого количества микроконтроллеров.

CAN (Controller Area Network) – технология промышленной сети, разработанная фирмой BOSCH для объединения в единую систему большого количества датчиков транспортного средства и бортового компьютера; используется преимущественно в мобильных системах, промышленной автоматизации и технологиях умного дома.

Режим передачи – последовательный, широковещательный, пакетный. Стандарт не описывает физический уровень, но чаще всего используется сеть с топологией шина на базе дифференциальной пары, стандарта ISO 11898. Передача ведётся кадрами, которые принимаются всеми узлами сети. Метод доступа к среде передачи данных с разрешением конфликтов (CSMA-CR) приоритетно обеспечивает доступ на передачу сообщения. Полезная информация в кадре состоит из идентификатора длиной 11 бит (стандартный формат) или 29 бит (расширенный формат) и поля данных длиной от 0 до 8 байт. Идентификатор говорит о содержимом пакета и служит для определения приоритета при попытке одновременной передачи несколькими узлами.

Особенностью CAN является высокая надежность передачи данных. Каждый передаваемый пакет защищен CRC32, а в случае потери пакета из-за помех производится автоматическое повторение передачи. Высокой надежности также способствует малый размер пакетов. CAN хорошо зарекомендовал себя и в настоящее время активно используется в автомобильной, промышленной и аэрокосмической технике.

Контроллер ЖКИ

Контроллер ЖКИ, встроенный в микроконтроллеры, предназначен для упрощения работы с сегментными дисплеями, требующими организации динамической индикации. Для зажигания одного сегмента необходимо записать активное значение (0 или 1) в соответствующие регистры, отвечающие за номер сегмента, и номер знакоместа. Временную диаграмму для нормального отображения знаков на ЖКИ контроллер вырабатывает самостоятельно. Фактически, контроллер ЖКИ представляет буфер видеопамяти и дает возможность сконфигурировать линии управления.

Интерфейс JTAG

Интерфейс IEEE 1149.1 JTAG предназначен для обеспечения тестирования устройства, отладки программ и загрузки программного обеспечения. Подробнее об этом интерфейсе можно прочитать в главе про отладку программного обеспечения.

Встроенный загрузчик

Встроенный загрузчик добавляется в микроконтроллер производителем, находится там всегда (его, как правило, невозможно уничтожить) и позволяет обновлять прошивку микроконтроллера через обычный последовательный канал, без использования дополнительного программного обеспечения со стороны целевой системы и аппаратного обеспечения (программатора) со стороны инструментальной.

Встроенный загрузчик находится в защищенной области встроенной FLASH памяти (ПЗУ), поэтому его нельзя стереть. Включение загрузчика производится путем подачи специального сигнала (или группы сигналов) на определенные выводы микроконтроллера или специального символа, сразу после старта микроконтроллера (часто так делается в микроконтроллерах FR50 фирмы Fujitsu). Например, в учебных стендах SDK-1.1 и SDK-2.0 встроенный загрузчик включается путем замыкания перемычки на плате.

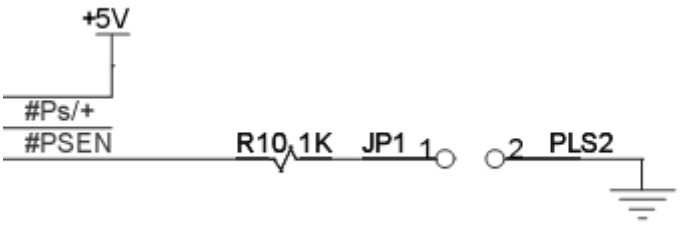


Рисунок 8. С помощью перемычки JP1 можно переключить ADuC 812 в режим программирования (учебный стенд SDK-1.1)

Для примера рассмотрим работу со встроенным загрузчиком учебного стенда SDK-1.1. Перед работой с загрузчиком необходимо замкнуть перемычку JP1. Это действие, как видно из схемы, изображенной на рис. 1, приводит к подаче логического нуля на вход контроллера PSEN (Program Store Enable). После рестарта управление передается уже не программе, а встроенному загрузчику.

Исходные тексты загрузчика со стороны ПК для Linux можно посмотреть в проекте GM3P на сайте http://embedded.ifmo.ru. В каталоге TOOLS есть файл gdownload.cpp.

Пользоваться этим загрузчиком очень просто. Рассмотрим пример скрипта для программы GM3P.

```
9600 openchannel /dev/ttyS0
  dl main.hex
bye
```

В первой строчке скрипта настраивается последовательный канал (скорость 9600, устройство /dev/ttyS0). Во второй строке вызывается команда dl с указанием HEX файла, который необходимо запрограммировать. В последней строке стоит команда bye, завершающая работу скрипта.

Необходимо заметить, что пользоваться встроенными загрузчиками не всегда удобно. К сожалению, не все загрузчики обеспечивают надежную доставку данных. Кроме того, встроенный загрузчик не решает проблем, связанных с доставкой данных в устройства, отличные от встроенной FLASH памяти микроконтроллера. Для решения этих проблем обычно разрабатывают более сложный загрузчик с большей функциональностью. Например, загрузчик UL, используемый в учебных стендах SDK-1.1. Исходные тексты загрузчика можно посмотреть на сайте http://embedded.ifmo.ru.

1.4. Варианты организации ПО ВВС

По способу организации программного кода при проектировании ВВС, можно выделить варианты реализации ПО в виде:

- монолитного кода;
- системы, состоящей из нескольких модулей (драйверы, прикладная программа);
- сочетания операционной системы реального времени (ОС РВ) и прикладной программы;
- ОС РВ, виртуальной машины, прикладной программы, работающей в рамках виртуальной машины (ПЛК, Embedded Java).

Как следует из названия, монолитная программная система является однородным куском кода, не имеющим четко выраженных частей. Достоинствами такой системы является простота организации внутренних связей и некоторая свобода в действиях программиста (особенно на начальных этапах разработки). Недостатков, к сожалению, значительно больше. Из-за свободы в связях между различными частями программы и полной видимости отдельных компонентов (переменных и функций) вероятность ошибок очень велика. Сложную монолитную систему нелегко построить, а разбить сложную задачу на части не представляется возможным. С увеличением размера монолитной системы ошибок становится все больше. На каком-то этапе своего развития количество ошибок и сложность изменения становятся такими, что дальнейшее развитие системы, ее использование и отладка практически невозможны.

Монолитные системы являются редким явлением в области встроенного

ПО. Как правило, такие системы делают новички, так как иначе не умеют, или многоопытные профессионалы, когда необходимо очень быстро сделать какой-либо не очень сложный и одноразовый прототип системы.

Модульные системы – распространенный вариант построения не очень сложных ВВС. Модульность на уровне исходных текстов или на уровне библиотек позволяет разделить систему на четко очерченные фрагменты, которые можно отдать для реализации, отладки и тестирования отдельным людям, а потом собрать вместе и проверить в сборе.

Проблемой таких систем является сложность. К сожалению, не все программисты обладают достаточно развитым системным мышлением и трудные участки кода могут реализовать не все. Нужен некий скелет системы, так называемый архитектурный шаблон, в английском языке называемый *framework*.

Для решения проблемы сложности обычно выделяют некоторое ядро, реализованное высококлассными разработчиками. Мы рассмотрим два варианта таких скелетов или архитектурных шаблонов: операционную систему реального времени (ОС РВ) и виртуальную машину.

ОС РВ в проектировании является некой постоянной составляющей, как бы вынесенной за скобки после анализа множества монолитных ИУС.

Что дает нам применение ОС РВ? Во-первых, ОС РВ – это средство распределения ресурсов между прикладными процессами, а также средство организации этих процессов. Во-вторых, ОС РВ – это отлаженный программный код. В-третьих, ОС РВ, как правило, является архитектурой с заранее известными плюсами и минусами. В четвертых, – это средство для организации связи с большой номенклатурой аппаратных средств (различных контроллеров, УВВ и т.п.). Самостоятельная поддержка множества протоколов обмена, различных процессоров и контроллеров часто оказывается нерентабельной для большинства компаний, занимающихся разработкой ИУС, и проще использовать готовые наработки.

Какие минусы может принести использование ОС РВ? Естественно, большинство ОС РВ имеющихся на рынке разрабатывалась как относительно универсальные системы. Универсальность, как правило, означает избыточность функций и, следовательно, необходимость в дополнительных аппаратных ресурсов для поддержки этих функций. Если в каком либо проекте используется готовая ОС РВ, есть возможность получения закрытой системы, т.е. системы со скрытой внутренней структурой. Против использования такого «черного ящика» есть много аргументов. Одним из них является невозможность или трудоемкость проверки системы (например, при сертификации) на отсутствие серьезных ошибок и разного рода неучтенного, «шпионского» программного кода.

В последнее время популярен способ проектирования систем на базе шаблонов. Так, в частности, в CoDesign проектах используют заготовки ОС РВ (планировщики, переключатели процессов, IPC и т.д.). Эти шаблоны используются на архитектурном этапе проектирования, а на выходе системы проектирования разработчик получает монолитный код. Такой подход лишен большинства недостатков, присущих использованию универсальных (или покупных) ОС РВ.

Итак, к причинам, заставляющим применять ОС РВ можно отнести:

- необходимость использования готовой, надежной и предсказуемой платформы (выделение из множества ПО некой стандартной составляющей (поддержка унификации, стандартизации, модульности));
- необходимость обеспечения параллельного функционирования прикладных процессов;
- необходимость обеспечения защиты процессов друг от друга;
- необходимость в готовых драйверах сети, УВВ (существует много сложных протоколов, сложных устройств, необходимо много человеко-лет для реализации).

Рассмотрим еще один способ борьбы со сложностью – виртуальную машину. Виртуальная машина позволяет изолировать все особенности аппаратуры и проблемы соблюдения реального масштаба времени от конечного программиста. Достигается это удобство дорогой ценой.

Во-первых, уменьшается производительность системы из-за необходимости исполнения виртуальной машины. Во-вторых, сужается свобода программирования.

Виртуальные машины очень активно используются в промышленности, в так называемых программируемых логических контроллерах. Языки для программирования контроллеров давно стандартизированы. Примером одного из таких стандартов является IEC-61131-3.

Ниже приведен пример программы, написанной на языке ST (структурный текст).

```
FUNCTION tu_no : DWORD
VAR
    result : DWORD;
END_VAR
    result := 3;
    IF pv_command = COMMAND_OFF THEN result := 1; END_IF;
    IF pv_command = COMMAND_VO  THEN result := 2; END_IF;
    IF pv_command = COMMAND_NO  THEN
        result := 3;
        light_night();
        check_20_min();
    END_IF;
    IF pv_command = COMMAND_TM  THEN result := 4; END_IF;
    IF pv_command = COMMAND_TK  THEN result := 5; END_IF;
    tu_no := result;
END_FUNCTION
```

Данная программа непроста для программистов, работающих с языками вроде Си или Си++. Больше всего язык ST похож на языки Ada или ALOGOL-68. С точки зрения программиста, к недостаткам ST можно отнести следующие:

- отсутствие гибкости (например, отсутствие указателей в языке);
- типизация, требующая явного преобразования типов;
- тяжеловесные языковые конструкции.

Перечисленные недостатки являются следствием повышения надежности процесса программирования при создании программных систем с повышенной надежностью.

2. Основные парадигмы и технологии программирования ПО систем реального времени

2.1. Стиль программирования

Стиль программирования – внутренне концептуально согласованная совокупность средств, базирующаяся на некоторой логике построения программ. Впервые этот термин введен в книге [10] "Основания программирования ".

Стиль программирования следует отличать от *стиля кодирования*, который сводится к определенному способу форматирования исходного текста. Термины *программирование* и *программа* необходимо определить более точно, чтобы не возникало неоднозначности. В настоящее время под программой понимается последовательность действий, совершаемых машиной Фон-Неймана, а в качестве стиля программирования рассматривается базирующийся на машине Фон-Неймана структурный стиль программирования. В таком контексте непонятно, можно ли считать программой текст, написанный на языке VHDL или Verilog, а также тексты, описывающие системы, работающие в других моделях вычислений.

Непейвода приводит следующую классификацию стилей программирования:

- сентенциальное программирование (Рефал, Prolog);
- функциональное программирование (Lisp);
- автоматное программирование;
- событийное программирование;
- структурное программирование (Си);
- параллельное программирование;
- объектно-ориентированное программирование [8, 9, 10].

Стиль программирования, модель вычислений, платформа

Стиль программирования находится над моделью вычислений, так как модель вычислений задает набор правил, в рамках которых реализуется тот или иной стиль. Между решаемой задачей, стилем программирования и моделью вычислений не должно быть концептуальных противоречий [10]. В противном случае возникает резкий скачок сложности проектируемой системы, увеличиваются количество ошибок, сроки выполнения, быстро заканчивается бюджет проекта. К сожалению, большинство программистов не воспринимает эту проблем, что обычно приводит к увеличению проектных бюджетов и огромному количеству ошибок.

В качестве примера нарушения гармонии между стилем программирования и моделью вычислений можно привести ОС РВ, в которых искусственно

создается потоковая модель вычислений, а в качестве стиля программирования используется структурное программирование (чаще всего на языке Си). Аналогичные проблемы есть в современных операционных системах общего назначения (Microsoft Windows 2000/XP/Vista, Linux, FreeBSD, Mac OS и т.д.) и в системах программирования для них (Java, C#, C++ и т.д.). В своей статье "Проблемы с потоками" [15] Эдвард Ли подробно освящает эту проблему.

Чтобы избежать противоречий, используют способ проектирования на базе *платформ*, в котором каждый слой имеет свою модель вычислений и гармонично связанный с ним стиль программирования. Каждый слой системы при этом является фундаментом для последующих слоев и перемешивания понятий, встречающегося в современных ОС РВ, не происходит. В качестве примера гармоничной системы можно привести проект Giotto, выполняемый в университете Беркли, США. Аналогичный подход используется и в нашей стране. В качестве примера можно привести систему "Терра", описанную в работе [5].

Концептуальные противоречия современной информатики

Ни одна формализация не является универсальной

Каждая формализация ограничена, и, более того, при достаточно серьезном анализе сама подсказывает собственные альтернативы (следствие теорем Геделя о неполноте и теоремы Тарского о невыразимости истины).

Даже если две формализации сложного естественного понятия формально не противоречат друг другу, они начнут противоречить и мешать при попытках более глубокого анализа либо развития системы (т. н. концептуальные противоречия). Даже явные противоречия упорно игнорируются и теорией, и практикой, зацикленной на позитивном мышлении, а концептуальные вообще не принимаются во внимание.

Поэтому соединять все мыслимые средства в одном и том же месте – верный путь к саморазваливающимся системам. Заметим, что такое соединение является общим местом в современной информатике, поскольку принято критиковать системы за то, чего в них нет. Возникающие при добавлении новых возможностей концептуальные противоречия никого не волнуют, пока не становится слишком поздно.

Пора рассматривать слова «универсальная система» как нечто подобное философскому камню. Иллюзия универсальности заставляет вводить в систему возможности, губительным образом расширяющие ее. Переход к следующему уровню понятий невозможен без запрещения многих методов, действовавших на предыдущем уровне. Но в программировании этому препятствует иллюзия универсальности, конкретизирующаяся в предрассудок совместимости, который заставляет при развитии системы тянуть за собой шлейф концептуально несовместимых устаревших понятий. Случаи, когда от этого предрассудка отказались, считаны [7].

Данный тезис становится понятным, если ясны концепции моделей вычислений и стилей программирования. Основным способом, решающим проблему концептуальной противоречивости систем, является разделение на слои, в каждом из которых реализуется свои модель вычислений и стиль программирования. Примером концептуально непротиворечивой системы можно считать знаменитую базовую многоуровневую модель открытых систем.

В качестве примера концептуального противоречия можно привести использование семафоров, блокирующих каналов и критических секций, управляющих процессами на уровне потоковой модели вычислений из прикладных процессов в современных операционных системах.

Гильбертовская концепция идеальных и реальных объектов и парадокс изобретателя

Большинство объектов и понятий в областях, пользующихся развитым математическим аппаратом, не имеют прямых интерпретаций в реальности. Но их устранение, даже если оно в некоторых случаях теоретически возможно, приводит к неприемлемому (как минимум, башня экспонент) разрастанию длины доказательств. Таким образом, окольные пути являются наиболее короткими и эффективными [7] .

Хорошим подтверждением данного тезиса является отсутствие формальных методик сквозного проектирования встроенных систем. Системы CoDesign, разрабатываемые с начала девяностых годов, практически не продвинулись в концептуальном плане. Единственное развитие таких систем заключается в коммерческом применении отдельных университетских разработок десятилетней давности.

Система оценки сложности математических понятий по их типам

Первый тип – объекты, второй тип – их множества, либо функции над объектами, третий тип – функции над объектами второго типа и т. д. Система знаний тем более необходима потому, что информатика неразрывно связана с современной экономикой, в которой индустрия обмана и самообмана (реклама) занимает решающее место, и то, что влияние этого же духа проникло в современную науку в виде системы грантов, которая требует безудержной саморекламы и мешает действительно серьезному теоретическому и методологическому осмыслению проблем. Это лишь частный случай вредного влияния на прогресс квазирелигии прогресса [7].

Действительно, в настоящий момент мы являемся свидетелями интересного процесса, происходящего в индустрии информационных технологий. Демонстрируемый нам средствами массовой информации прогресс носит скорее технологический, чем концептуальный характер. Совершенствуется способ (технология) производства, позволяющий выпускать

продукты более высокими тиражами за меньшие деньги. При всем этом видимом прогрессе, начинка изделий остается на уровне понимания середины прошлого века. Можно также привести пример из автомобильной промышленности. В современных автомобилях за десятки лет производства меняется кардинально только вид корпуса и технология производства. Концепция двигателя внутреннего сгорания при всей своей архаичности останется неизменной до тех пор, пока выгодно производство и есть нефтяные запасы для изготовления топлива.

Состояние дел искусно маскируется и ретушируется крупными производителями, а в университетах дается весьма искаженное представление реального положения дел.

Игнорирование отрицательных результатов (позитивное мышление)

Игнорирование отрицательных результатов или, в более общем виде, позитивное мышление и квазирелигия прогресса. Закрывание глаз на теоретические предупреждения никогда ни к чему хорошему не приводит, а их систематическое изгнание из учебных курсов в угоду душевному комфорту и «позитивности» вредно сказывается на способностях к творческому мышлению, один из приемов которого: превратить минус в плюс[7].

Фактически, этот тезис покрывает всё области программирования. Например, отсутствие возможности протестировать встроенную систему из-за физической невозможности проверить все возможные ветвления алгоритма воспринимается как само-собой разумеющееся и в результате мы имеем множество потенциально ненадежных систем.

Дурно понимаемый прагматизм

Из теории выхватывается что-либо одно, и как можно быстрее оформляется в виде практической системы. Наблюдение за состоянием дел в современном “искусственном интеллекте” и информатике приводит к следующему закону экологии искусственных систем.

- Первая система, декларировавшая успех в данной экологической нише, захватывает ее и изгаживает таким образом, чтобы никакая другая в ней не могла выжить.
- Еще одной стороной того же прагматизма является ранняя стандартизация. Случайные особенности недозрелой системы фиксируются в стандарте, и, поскольку лучшие специалисты в данной области слишком часто имеют хакерские наклонности и привыкли использовать недоделки в качестве якобы виртуозных приемов, эти недоделки возводятся в ранг священной коровы. Например, такой статус приобрела ошибка в реализации алгоритма унификации в раннем Prolog и конкретный способ реализации возвратов, который был там применен.

> Здесь нужно было бы в позитивном смысле воспринять Оруэлла, у которого словарь Новояза вплоть до 11 издания был предварительным, поскольку продолжались усилия обеспечить концептуальное единство языка и убрать лишние возможности[7].

В области встроенных систем в качестве дурно понимаемого прагматизма можно привести пример широкого внедрения операционных систем реального времени, которые таковыми на самом деле никогда не являлись. Если посмотреть научные статьи, посвященные проблемам алгоритмов планирования, то становится ясно, что задача получения реального масштаба времени решена только для достаточно узкого класса задач и работает при сильных ограничениях (таких^ например, как отказ от средств IPC). Ни один из производителей коммерческих ОС РВ не заявляет о вышеозначенных проблемах.

Программирование больше похоже на стихи, чем на прозу

Без самоограничения не достигнешь совершенства, и во всех случаях, когда пытались обеспечить программистам «свободу выражения», это приводило лишь к концептуальным противоречиям и открывало настежь двери для хакерства. Так что нужно нацеливать мысль, а не освобождать ее. Свобода для творца не является ценностью, если продуктами его творения будут вынуждены пользоваться другие. [7]

Это очень важный тезис, показывающий необходимость границ, даваемых архитектурным шаблоном (по англ. *framework*). Если программист не в состоянии придумать свой архитектурный шаблон, то ему нельзя ни в коем случае давать возможность решать задачу по своему, так как без понимания концепции получится полная анархия, приводящая к концептуальным противоречиям.

Если обратить внимание, то мы окружены подобными архитектурными шаблонами. Они, как правило, опробованы большим количеством людей и имеют известные свойства.

Стиль, метод, методология, методика

Стиль программирования реализуется через методологии программирования, заключающиеся в совокупности соглашений о том, какие базовые концепции языков программирования и их сочетания считаются приемлемыми или неприемлемыми для данного стиля. Методология включает в себя модель вычислителя для данного стиля.

Методология реализуется через методики, которые состоят из следующих компонент.

- Поощрение (или прямое предписание) использования некоторых базовых

концепций программирования.

- Запрещение (или ограничение) применения некоторых других базовых концепций. Иногда запрещение либо ограничение может быть неявным, через исключение нежелательных концепций из предписываемого языка или его диалекта.
- Требования и рекомендации по оформлению и документированию программ.
- Совокупность инструментальных и организационных средств, поддерживающих все вышеперечисленные требования и рекомендации.[9]

Теоретические исследования последних десятилетий показали, что различным классам задач и методов соответствуют различные логики построения программ, и эти логики несовместимы друг с другом. Логики (и, соответственно, стили программирования) дают самую общую оболочку, не зависящую от конкретных предметных областей и даже от конкретных методов. Как только мы уясняем себе особенности методов, стили начинают конкретизироваться далее. Так, известный (поскольку именно он преподается в качестве единственно существующего во всех современных учебных пособиях по началам программирования) структурный стиль конкретизируется в циклический либо рекурсивный варианты (*ипостаси*).

В каждом стиле имеются *ипостаси*. **Ипостаси** – формы, в которых высокоуровневое и абстрактное понятие стиля проявляется в наших конкретных построениях. Ипостаси логически друг другу не противоречат, но фактически непримиримо враждуют (порою даже больше, чем разные стили), если пытаться использовать их вперемежку. Это связано с тем, что ипостаси настроены на разные дисциплины расходования ресурсов.

В структурном стиле имеется две ипостаси: циклическая и рекурсивная. Циклическая ипостась хороша, когда структуры данных статические и не слишком глубокие, и очередной слой информации практически полностью используется следующим слоем. Рекурсивная ипостась используется, когда структуры данных динамические и глубокие, и на следующем слое нужна лишь малая часть предыдущего слоя, либо слои сами по себе имеют малую ширину.

В автоматном стиле имеется две ипостаси: последовательная и параллельная. Последовательная ипостась хороша, когда система представляется как единый блок информации, а параллельная – когда система делится на относительно автономные взаимодействующие подсистемы.

Заметим, что на автоматном стиле можно проследить, как одни и те же программы можно реализовать разными методами программирования, лежащими внутри одного и того же стиля. Методы реализации близки к технологической дисциплине, они мешают друг другу при произвольном смешении скорее технологически, чем логически или ресурсно. Если

программа построена эклектической смесью нескольких методов, ее труднее отлаживать и особенно модифицировать.

В событийном стиле имеются две ипостаси: от событий и от приоритетов. Здесь можно увидеть, что стиль появляется и существует независимо от языковой поддержки и порою вопреки ей.

В функциональном стиле ипостаси пока не появились, поскольку он сам по себе не был осознан и все время смешивается с рекурсивной ипостасью структурного программирования. Здесь можно увидеть, как недостаток внимания к интерфейсам между разными стилями приводит к интеграции одного стиля в другой. Даже когда (как в данном случае) они неплохо взаимодействуют, они еще лучше существовали бы по отдельности. [10]

Практические рекомендации по использованию стилей программирования

В книге [10] приведен ряд советов по использованию стилей программирования в системах общего назначения. Мы попытаемся выбрать советы, применимые в области встроенного ПО. Хочется заметить, что советы обычно имеют частный характер, и не стоит их возводить в ранг закона и рассматривать как истину в последней инстанции.

1. Если у Вас появляется много структурных или неструктурных переходов, либо все время приходится присваивать значения признакам, а затем их проверять, посмотрите, нельзя ли выделить модуль в автоматном стиле.
2. Если у Вас появляется много структурных или неструктурных переходов, либо все время приходится присваивать значения признакам, а затем их проверять, посмотрите, нельзя ли выделить модуль в автоматном стиле.
3. Если Вы начинаете думать о задаче как о вычислении (не обязательно над числами), пользуйтесь структурным программированием.
4. Если Вы намерены переложить понравившуюся Вам нечисленную (например, алгебраическую, топологическую или логическую) теорему в алгоритм для решения Вашей задачи, подумайте о функциональном стиле.
5. Если у Вас каждое следующее значение требует немногих данных, но различные значения обращаются к одним и тем же, даже не пытайтесь программировать рекурсивно, рекурсивное описание можете сохранить в качестве документации к программе, а саму программу пишите в терминах циклов и массивов.
6. Если данные строго подразделяются на ветви, используемые разными последующими значениями, лучше писать рекурсивно.
7. Если Вы смотрите на данные как на активные единицы, которые взаимодействуют между собой, воспользуйтесь объектно-ориентированным программированием.
8. Если в предыдущем случае объекты в том виде, в котором они

предоставляются современными системами программирования не подошли, воспользуйтесь каким-либо пакетом для организации квазипараллельной работы в условном времени и программируйте от событий. [10].

Необходимо заметить, что применение автоматного стиля программирования порождает весьма громоздкий код при реализации задач, явно требующих параллелизма. Для упрощения задачи можно использовать ОС РВ для реализации многозадачности. Практика показывает, что в области встроенных систем автоматное программирование чаще всего встречается при реализации сетевых протоколов. Если стек протоколов хорошо раскладывается на уровни, которые при этом не перемешаны, задача достаточно просто решается в рамках модели сетей процессов Кана (подробнее об этом можно узнать в следующем разделе). Каждый уровень протокола реализуется в виде бесконечного цикла, принимающего данные из одной очереди FIFO и передающего результаты в другую очередь FIFO. Псевдопараллельное исполнение каждого такого цикла реализуется за счет переключателя задач.

Сложность реализации такой задачи в рамках автоматного подхода объясняется тем, что мы пытаемся реализовать модель вычислений, не соответствующую выбранному стилю программирования.

Непейвода в книге [10] приводит также несколько практических рекомендаций о сочетании различных стилей программирования. В частности, он считает, что в сравнительно больших (более 500 строк исходного текста) программах есть место различным стилям программирования. Попытаемся выбрать рекомендации по объединению стилей, подходящие для встроенного ПО.

1. Структурное и автоматное программирование нужно как можно жестче разделять на уровне модулей. Вопрос о языковой совместимости и интерфейсах здесь не возникает, поскольку оба они реализуются стандартными средствами традиционных языков.
2. Две ипостаси структурного программирования также следует жестко разделять средствами модульности.
3. Параллельная ипостась автоматного программирования в настоящий момент может быть реализована практически только последовательными средствами, воспользуйтесь системой моделирования в условном времени.
4. Событийное программирование стоит выделять в самостоятельные модули.
5. В настоящий момент имеется целый ряд языков, основанных на идеях функционального программирования и продолжающих тенденции языка LISP на более современном уровне. Они имеют интерфейсы с C++ и Java, так что для написания модуля в функциональном стиле внутри большой традиционной программы лучше воспользоваться Ocaml или Haskell. [10]

2.2. Модель вычислений

Модель вычислений, вычислительная модель (model of computation, MOC) – набор законов взаимодействия элементов вычислительной системы.

Модель вычислений – набор правил организации вычислительного процесса, в рамках которых возможен его формальный анализ.

Модель вычислений – набор формальных правил, в рамках которых организована взаимосвязь и поведение множества составляющих атомарных частей модели некоторой вычислительной системы.

Модель вычислений – строго определенная парадигма (набор правил), описывающая протекание вычислительного процесса, способы обмена данными, взаимодействия между отдельными функциональными элементами.

Модель вычислений – недвусмысленный формализм для представления спецификаций проекта и проектных решений.

Модель вычислений – математическая модель, демонстрирующая пользователю вычислительные возможности вычислителя и правила их использования.

Введение в модели вычислений

Существует реальная, физическая вселенная. То, что обычно называют материей или суперпозицией неких частиц и полей. Мы не знаем, как и из чего она сделана, мы можем только догадываться. Существует множество теорий, *пытающихся* объяснить строение материи, но это только теории, они верны, пока не опровергнуты какими либо новыми экспериментами, объясняющими, в свою очередь, новые теории. Теории эти позволяют нам лишь увидеть упрощенные модели *реальной* вселенной.

В процессе изготовления какой-либо вещи мы используем реальную материю, руководствуясь текущей моделью того, что она собой представляет. Атомов, электронов, позитронов и кварков скорее всего на самом деле нет, это абстракции, придуманные физиками, вроде монад Лейбница (неких неделимых элементов материи). Абстракции позволяют нам упростить и как-то понять сверхсложную материю реальной вселенной.

Вселенной свойственны некоторые физические законы. Каждый уровень абстракции – это отпечаток или модель вселенной. На каждом уровне абстракции тоже есть свои законы. Можно сказать, что каждый уровень абстракции (струнный, элементарных частиц, атомарный, транзисторный, вентильный, модульный и т.п.) является некоторой вселенной, также имеющей свои законы. И законы эти, как показывает практика, – разные, и вселенные (модели) существуют параллельно.

Зачем представлять реальный объект, в нашем случае вычислительную систему, как множество моделей? Ответ прост. При анализе системы мы не можем охватить всю ее целиком на самом низком уровне абстракций (например, кварковом). Получится слишком сложно для понимания из-за большого количества мелких деталей. Человеческий мозг просто не в состоянии переварить такое количество информации.

Таким образом, толщина слоев между моделями определяется следующими вещами:

- обозримостью или прозрачностью для восприятия человеком;
- наличием подходящей теории (языка и законов функционирования) для описания очередного (промежуточного) слоя/модели.

Далее, мы замечаем, что одна модель не в состоянии описать весь уровень абстракции, чтобы мы могли с уверенностью заявить "да, этот уровень понятен, можно опускаться ниже". В результате, на одном уровне абстракции оказывается целый пакет моделей (срез, архитектурная модель, гештальт).

Принимая к сведению, что в моделях последовательно, слой за слоем раскрывающих нам внутреннее устройство вычислительной системы, действуют разные законы функционирования, мы можем заявить, что одни и те же термины могут описывать разные вещи. Кроме того, в разных моделях используются различные формы представления. В некоторых созданных нами вселенных нет места для времени (структурная схема), а в некоторых – есть (временная диаграмма, диаграмма взаимодействий). В некоторых нет места для причинно-следственных связей (сети процессов Кана), а в некоторых – есть (FSM). Между моделями одного пакета связь существует, но она достаточно тонка. Одна модель раскрывает какой-либо один (или несколько) аспектов системы. Весь пакет раскрывает все. Явной связи, способа трансформации из одного в другое нет, так как это описание разных частей целого. Попробуйте из формы батареи отопления вывести химический состав стенки в комнате или необходимую толщину кирпичей формальным способом.

Обычно отнесение строительных блоков проектируемых систем к той или иной категории вызывает определенные проблемы у начинающих проектировщиков, что обычно порождает серьезные проблемы архитектурного уровня.

Взаимосвязь понятий: система, элементная база, платформа, уровень абстракции, модель вычислений

Для того, чтобы понять, что такое элементная база, что является элементами этой базы и по какому принципу можно эти элементы классифицировать, нам необходимо уяснить суть таких часто употребляемых в вычислительной технике терминов как платформа, уровень абстракции, модель вычислений, структура, поведение, функциональность и архитектура системы.

Перечисленные термины окружают элементную базу со всех сторон и, не учитывая их, понять, что это такое будет достаточно сложно. Обычно, специалисты, услышав один из этих терминов, на вопрос "что это такое?" отвечают либо «не знаю», либо «это и так всем понятно». Однако элементарный поиск в Интернете или в литературе показал, что не так все просто, как кажется.

Система

Итак, начнем с системы. Обычно системой мы называем то, что проектируем, особо не задумываясь над смыслом термина, подразумевая нечто сложное, составное. Из теории систем нам известно определение – «система есть организованная, упорядоченная совокупность объектов и взаимосвязей между ними2. Из этого определения понятно, что есть некоторое множество объектов, составляющих нашу систему и какие-то разумные связи между объектами. Вроде бы все просто, и даже элементарно, но почему же наша система, которую мы создаем, никак не отлаживается? Что за объекты такие, из которых состоит наша система? Почему именно они попали в систему, а не иные?

Элементная база

Объекты – наш строительный материал, элементная база. Как определить, что этот объект подходит для строительства нашей системы? Начнем с примеров. Попробуем представить строительство обычного жилого дома. Строители выделяют нужные им объекты достаточно легко: для построения пола и потолка они берут бетонные плиты, для стен – кирпичи и раствор, в оконные проёмы они вставляют окна, а в дверные проёмы – двери. Для строителей кирпичи, бетонные блоки, окна и двери являются строительными материалами, а с нашей позиции – элементной базой. Отметим, что в перечень строительных материалов (элементной базы) нам (и тем более профессиональному строителю) уже на подсознательном уровне не добавить холодильник, кусок пиццы, самолёт или собаку. Мы эти потенциальные объекты для стройки считаем чуждыми и сразу отбрасываем. Почему? Как понять, что является элементной базой, а что нет? Как отфильтровать подходящее для строительства от негодного, по какому критерию?

Первая мысль,которая нам приходит в голову – «они стыкуются друг с другом, как в конструкторе Lego». Да, это так. Все строительные материалы сделаны так, чтобы их можно было соединить определенным способом. Для разных видов строительства есть разные наборы исходных элементов, причем, как правило, с конкурирующими наборами они сочетаются неважно. Аналогичная ситуация существует и в вычислительной технике: микросхемы, принадлежащие разным производителям, плохо стыкуются друг с другом, несмотря на схожую функциональность. В данном случае можно говорить о принадлежности элементов к некой группе. Для того, чтобы система

заработала, придется собрать составные части из одной группы. Получается, что одно и тоже решение можно получить несколькими способами.

Попробуем сформировать определение термина элементная база: *набор объектов, объединенных общими принципами взаимодействия*. Вроде как хорошо, но чего-то не хватает. Интересно, а почему цемент и песок не попали в список? Это ведь тоже строительный материал. Ответ прост: из этих элементов делают строительный раствор и они находятся на другом, более низком *уровне абстракции.*

Уровень абстракции

Что такое уровень абстракции? Первое, что нам обычно попадается при поиске в интернет – Википедия. Из английской версии Википедии мы узнаем, что абстракция – это способ сокрытия деталей реализации определенной функциональности. Далее в статье приводится семиуровневая модель OSI/ISO в качестве яркого примера. Кстати, если хотите, чтобы студент не сдал экзамен, попросите его объяснить, что означают эти уровни в модели.

Так неужели авторы этой знаменитой модели применили уровни абстракции, чтобы просто скрыть реализацию? Едва ли. Было бы забавно посмотреть на рабочих, скрывающих реализацию функциональности цементного раствора.

Реализация, конечно, не видна, это факт. Это справедливо и для модели OSI/ISO, и для цементного раствора. Но в данном случае сокрытие реализации скорее не причина, а следствие.

Модель вычислений и уровень абстракции

Элементы системы, находящиеся на одном уровне абстракции, довольно хорошо стыкуются друг с другом. Получается, что уровень абстракции не скрывает ничего специально, а просто объединяет в себе элементы со схожими свойствами. Если точнее, то уровень абстракции – это такое место, где существуют определенные законы существования элементов, можно сказать,в селенная с отдельными законами. Вы когда-нибудь видели параллельную вселенную? Нет? О ней никто ничего не знает, кроме писателей фантастов, ее никто не видел и ученые на эту тему пока молчат. Вот и соседний уровень абстракции закрыт от нас так же как эта параллельная вселенная.

Попробуем сформулировать определение по новому: **уровень абстракции** *системы – этап формирования системы с заданным набором законов взаимодействия элементов*.

Вернемся от строительного примера обратно в вычислительную технику. О каких законах взаимодействия идет речь? Речь идет о достаточно сложном понятии вычислительной техники – **модели вычислений**. Попробуем дать

определение: **модель вычислений** – *это набор законов взаимодействия элементов вычислительной системы*. Получается, что модель вычислений существует в рамках уровня абстракции и определяет способы функционирования элементной базы, из которой мы собираемся строить систему.

Платформа

Рассмотрим подробнее уровни абстракции. Правильно ли мы трактуем модель, например, семиуровневую OSI/ISO как слоёный пирог? Сравнение не совсем корректное. Разберемся поподробнее. Каждый слой имеет свои законы функционирования, т.е. в нем действует своя модель вычислений. Каждый слой является основной **платформой** для более высокого слоя. Он создает условия для его существования, как, например, физический уровень Ethernet создает условия для существования канального уровня. Какие тут параллели из жизни можно провести? Хорошим примером будут компьютерные игры в стиле Action. Представьте, что на своем компьютере вы запускаете какой-нибудь Quake или Unreal. Вы наблюдаете на экране монитора виртуальную вселенную. Персонажи этой вселенной вас не видят. Виртуальная вселенная компьютерной игры целиком и полностью зависит от игровой виртуальной машины (так называемого «движка»), последний, в свою очередь, от операционной системы, а она – от персонального компьютера и так далее. Получается картина не пирога, а скорее вложенных как матрёшки друг в друга не пересекающихся вселенных.

В ранних публикациях Intel, кстати, очень любили графически представлять системы в виде концентрических окружностей: в центре процессор, далее базовая система ввод-вывода (BIOS), операционная система и внешний слой – прикладная программа.

Попробуем теперь сформулировать определение для *платформы*. Платформа нам обычно представляется чем-то незыблемым, твердым, надежным. Платформа – это то, на чем можно стоять. Фундамент дома, например. Является ли платформа чем-то законченным? Обычно нет, если эта платформа не железнодорожная. Как правило, платформа – это законченный этап строительства, эволюции или развития системы. Итак, **платформа** – *очередной, законченный этап в эволюции системы*.

Архитектура

Осталось теперь понять суть термина **архитектура**. Как правило, его считают синонимом термина структура. Отчасти это верно. Есть определение, гласящее, что архитектура есть концептуальная модель, то есть описание концепции, сути работы системы. Если мы изложим структуру, то покажем только один из уровней абстракции нашей системы. Если уровней абстракции

только один, то архитектура и структура сливаются. Например, для простейшей студенческой программы на языке Pascal будет достаточно блок схемы алгоритма. А вот если уровней много, тогда для описания всей концепции работы системы нужно сделать описание нескольких структур, для каждого из уровней. До какого уровня нужно описывать систему, чтобы рассказать о ее работе достаточно полно с одной стороны и не перегружать описание ненужными деталями с другой? Ответ достаточно прост. Необходимо затронуть все слои разрабатываемой нами системы и остановиться на твердом основании, то есть на платформе. Таким образом у нас получается, что а*рхитектура системы* – это совокупность структур, описывающих систему до уровня платформы.

Основные идеи, лежащие в основе концепции "модель вычислений"

Ключевой задачей в процессе проектирования является обоснованный выбор определенной модели целевой системы. Рассмотрение системы с точки зрения той или иной модели автоматически привнесет в систему свойства, присущие выбранной модели.

Обычно при рассмотрении модели встроенной системы говорят о так называемой модели вычислений. Модель вычислений можно представлять как строго определенную парадигму (набор правил), описывающую протекание вычислительного процесса, способы обмена данными, взаимодействия между отдельными функциональными элементами. Кроме того, модель вычислений предлагает терминологию и примитивы (элементную базу), в базисе которых требуется выражать и описывать целевую систему. Модель вычислений описывает природу потоков данных, элементов синхронизации, роль времени в процессе выполнения системой целевой функции. Различные модели вычислений по-разному описывают одни и те же процессы, протекающие в целевой системе. Для больших и сложных систем совершенно нормально положение дел, когда различные части системы представляются различными моделями вычислений.

Язык модели вычислений или стиль программирования

Важной и неотъемлемой частью модели вычислений является язык модели вычислений или стиль программирования, в терминологии Н.Н. Непейводы. Как и любой другой язык, язык модели вычислений определяется алфавитом, синтаксисом и семантикой. Алфавит представляет собой множество допустимых символов языка, которые могут быть скомбинированы различными способами. Правила комбинирования и допустимые комбинации определяются синтаксисом языка. Смысл и интерпретация тех или иных допустимых комбинаций символов алфавита определяется семантикой языка. С точки зрения языка его модель вычислений представляет собой поведение некоторой абстрактной вычислительной машины (или просто абстрактного вычислителя)

в рамках семантики языка. С этой точки зрения можно вести разговор о моделях вычислений традиционных языков программирования, таких как Си, C++, Java, Pascal и т.д. В этом случае, оценивая применимость того или иного языка, а на самом деле модели вычислений, на которой каждый конкретный язык базируется, для решения той или иной задачи, стоящей перед разработчиком, можно обсуждать и оценивать характеристики системы, которые навязывает МВ.

Разработка встроенных систем с использованием модели вычислений

Традиционно в процессе проектирования, практически на всех уровнях абстракции, система рассматривается как набор изолированных компонентов (черных ящиков) с определенной функциональностью. Выполняя свои функции в конкретные моменты времени компоненты «поглощают» входные данные и «производят» выходные данные. Этими данными компоненты обмениваются посредством некой коммуникационной среды. На более абстрактных уровнях рассмотрения системы относительно текущего многие компоненты текущего уровня с коммуникациями превращаются в новые «черные» ящики, скрывая реализацию. На более конкретных уровнях рассмотрения системы относительно текущего становится более конкретной реализация каждого черного ящика, видны составляющие его блоки и коммуникации между ними. Ценность и удобство модели вычислений на определенном уровне абстракции заключается в том, чтобы она не была слишком абстрактной или подробной. В первом случае модель будет не в состоянии описать процессы, происходящие в целевой системе, с интересующей разработчика точки зрения. Во втором случае для составления полной модели необходим будет такой объем данных, что сложность представления системы окажется неадекватной и бесполезной.

Если рассмотреть целевую систему на некотором уровне абстракции как набор взаимодействующих изолированных блоков (вычислительных компонентов системы), то модель вычислений системы на данном уровне абстракции, используя выразительные средства языка, описывает следующие аспекты системы:

- поведение вычислительных компонентов;
- взаимодействие вычислительных компонентов;
- способы передачи данных и синхронизацию вычислений;
- способы декомпозиции и агрегации вычислительных компонентов.

В настоящее время предлагается множество моделей вычислений встроенных систем, обладающих определенными свойствами и имеющих свои достоинства и недостатки. Для каждой из моделей вычислений необходимо учесть следующие моменты:

- адекватность описания целевой системы;

- удобство моделирования;
- формальность методов реализации и верификации.

Модель вычислений должна содержать характеристики системы, важные на данном уровне абстракции. Элементы модели (примитивы, языковые средства, требования и др.), а, следовательно, и вся модель в целом, не должны быть слишком абстрактными или слишком конкретными. Т.е. модель вычислений должна быть в состоянии описать целевую систему. Описание в данном контексте понимается как определенная степень спецификации системы, возможно не до конца формальной. Это требование к модели вычислений можно сформулировать как адекватность описания целевой системы.

Кроме описания целевой системы на заданном уровне абстракции, модель вычислений должна обеспечивать разработчику средства работы с этим описанием. Разработчик должен иметь возможность доказывать истинность или ложность определенных утверждений относительно целевой системы, проверять соответствие определенным требованиям и ограничениям, накладываемым на целевую систему. Инструменты, предоставляемые моделью, должны позволять проводить оценку тех или иных характеристик целевой системы, оптимизацию по выбранным параметрам. Все эти действия разработчика можно назвать моделированием целевой системы в терминах выбранной модели вычислений.

После обработки и корректировки модели на данном уровне абстракции разработчик должен повышать «конкретность» модели, т.е. реализовывать модель. В процессе реализации могут потребоваться значительные ресурсы, чтобы, оставаясь в ограничениях текущей модели вычислений, повысить конкретику представления целевой системы, возможно с переходом на другую модель вычислений. Зачастую требуются достаточно формальные методы трансформации моделей вычислений, чтобы в конечном итоге у разработчика оставалась возможность проведения эквивалентной верификации получаемых моделей. Необходимы формальные методы трансформации требований, ограничений, спецификаций, функциональности вычислительных компонентов и коммуникаций между ними. Чем более формальные методы описанных процессов предлагает модель вычислений, тем удобнее с ней работать, и результаты такой работы становятся более предсказуемыми.

2.3. Принцип «KISS»

Принцип «KISS» (англ. Keep It Simple, Stupid – «будь проще, тупица») – процесс и принцип проектирования, при котором простота системы декларируется в качестве основной цели и/или ценности. За многие годы

использовались разные расшифровки акронима KISS, и есть некоторые сомнения в том, которая из них является оригинальной.

Эта концепция имеет прямую аналогию с «бритвой Оккама», а также с утверждением Альберта Эйнштейна, что «*всё должно быть сделано настолько простым, насколько это возможно, но не проще*»[23].

Приведенные ниже рецепты, несмотря на свой неформальный, эмпирический характер имеют достаточно большую ценность, так как были сформулированы весьма известными инженерами, работающими в области информационных технологий. Необходимо заметить, что данный материал нужно принимать с известной долей скептицизма, так как предлагаемые решения носят частный характер.

Дуг МакИлрой

Дуг МакИлрой, изобретатель каналов UNIX и один из основателей традиции UNIX, обобщил философию следующим образом.

«Философия UNIX гласит.

1. Пишите программы, которые делают одну вещь и делают её хорошо.
2. Пишите программы, которые бы работали вместе.
3. Пишите программы, которые бы поддерживали текстовые потоки, поскольку это универсальный интерфейс».

Обычно эти высказывания сводятся к одному «делайте одну вещь, но делайте её хорошо». Из этих трёх принципов только третий является специфичным для UNIX, хотя разработчики UNIX чаще других акцентируют внимание на всех трёх принципах[1].

Пайк: стиль программирования на C

Роб Пайк (англ. Rob Pike) предложил следующие «правила». Стиль программирования на C в качестве аксиом программирования. Одновременно эти правила могут выражать точку зрения на философию UNIX.

Правило 1. Вы не знаете, где программа начнет тормозить. Узкие места возникают в неожиданных местах, поэтому не стройте догадки и изучайте скорость работы программы до тех пор, пока не удостоверитесь, что узкое место найдено.

Правило 2. Измерение. Не оптимизируйте скорость до тех пор, пока ее не

измерите, и даже если вы проверили какую-то часть кода с узким местом, проверьте остальные.

Правило 3. Изощрённые алгоритмы являются медленными, если n мало, а n обычно мало. В изощрённых алгоритмах присутствуют большие константы. До тех пор, пока вы не убедитесь, что n часто становится большим, избегайте изощрённости. (Даже если n становится большим, вначале используйте правило 2).

Правило 4. Изощрённые алгоритмы чаще подвержены ошибкам, чем их простые аналоги, также их гораздо сложнее реализовать. Используйте простые алгоритмы наряду с использованием простых структур данных.

Правило 5. Данные преобладают. При правильной и хорошо организованной структуре данных, алгоритмы становятся очевидными. Структуры данных, а не алгоритмы, являются центральной частью в программировании.

Правило 6. Правила 6 нет.

Правила 1 и 2 Пайка перефразированы Тони Хоаром (Tony Hoare) в известную аксиому «Преждевременная оптимизация — корень всех зол». Кен Томпсон (Ken Thompson) перефразировал правила 3 и 4 Пайка так: «Если сомневаетесь, используйте перебор всех возможных комбинаций». Правила 3 и 4 являются частными положениями философии дизайна KISS: Keep It Simple, Stupid (будь попроще, тупица). Правило 5 было предварительно сформулировано Фредом Бруксом (Fred Brooks) в книге «Мифический человеко-месяц». Правило 5 часто сокращают до «пиши тупой код, который использует умные данные». Правило 6 взято из шутки Брюса, прозвучавшей в «Летающем цирке Монти Пайтона».

Майк Ганцарз: философия UNIX

В 1994 году Майк Ганцарз (Mike Gancarz) объединил свой опыт работы в UNIX (он является членом команды по разработке системы X Window) с высказываниями из прений, в которых он участвовал со своими приятелями программистами и людьми из других областей деятельности, так или иначе зависящих от UNIX, для создания Философии UNIX, которая сводится к 9 основным принципам.

1. Маленькое прекрасно.
2. Пусть каждая программа делает одну вещь, но хорошо.
3. Собирайте прототип как можно раньше.
4. Предпочитайте переносимость эффективности.
5. Храните данные в простых текстовых файлах.
6. Используйте программные рычаги для достижения цели.

7. Используйте сценарии командной строки для улучшения функционала и переносимости.
8. Избегайте связывающего программу (captive) пользовательского интерфейса.
9. Делайте каждую программу «фильтром».

Менее важные 10 принципов не снискали всеобщего признания в качестве частей философии UNIX и в некоторых случаях являлись предметом горячих споров (Монолитное ядро против Микроядра).

1. Позвольте пользователю настраивать окружение.
2. Делайте ядра операционной системы маленькими и легковесными.
3. Используйте нижний регистр и придерживайтесь кратких названий.
4. Храните данные древовидно.
5. Молчание — золото.
6. Думайте о параллельности.
7. Объединенные части целого есть нечто большее, чем просто их сумма.
8. Ищите 90-процентное решение.
9. Лучшее – враг хорошего.
10. Думайте иерархически.

Реймонд: искусство программирования в UNIX

Эрик С. Рэймонд (Eric S. Raymond) в своей книге «Искусство программирования в UNIX» подытожил философию UNIX как широко используемую инженерную философию «Будь попроще, тупица» (принцип KISS). Затем он описал, как эта обобщенная философия применима в качестве культурных норм UNIX. И это несмотря на то, что несложно найти несколько нарушений в следующей текущей философии UNIX:

Правило Модульности: пишите простые части, соединяемые понятными интерфейсами.

Правило Ясности: ясность лучше заумности.

Правило Композиции: разрабатывайте программы так, чтобы их можно было соединить с другими программами.

Правило Разделения: отделяйте правила (policy) от механизма (mechanism); отделяйте интерфейс от движка (engine).

Правило Простоты: нацельтесь на простоту; добавляйте сложность, только там, где необходимо.

Правило Экономности: пишите большую программу только когда можно продемонстировать, что другими средствами выполнить необходимую задачу не удастся.

Правило Прозрачности: разрабатывайте прозрачные программы для

облегчения последующего пересмотра и отладки.

Правило Надёжности: надёжность — дитя прозрачности и простоты.

Правило Представления: храните знания в данных так, чтобы логика программы была тупой и надёжной.

Правило Наименьшего удивления: при разработке интерфейса всегда делайте как можно меньше неожиданных вещей.

Правило Тишины: если программе нечего сказать, пусть лучше молчит.

Правило Восстановления: если надо выйти из строя, делайте это шумно и как можно быстрее.

Правило Экономии: время программиста дорого; сократите его, используя машинное время.

Правило Генерации: избегайте ручного набора кода; при любом удобном случае пишите программы, которые бы писали программы.

Правило Оптимизации: сначала – опытный образец, потом – «причесывание». Добейтесь стабильной работы, только потом оптимизируйте.

Правило Многообразия: отвергайте все утверждения об «единственно правильном пути».

Правило Расширяемости: разрабатывайте для будущего. Оно наступит быстрее, чем вы думаете.

Большинство из этих норм принимается вне сообщества UNIX, даже если это было не так во времена, когда они впервые были применены в UNIX, то впоследствии это стало так. К тому же много правил не являются уникальными или оригинальными для сообщества UNIX. Тем не менее, приверженцы программирования в UNIX склоняются к тому, чтобы принять комбинацию этих идей в качестве основ для стиля UNIX[1],[4].

Джерард Хольцман: 10 правил разработки программ с особыми требованиями к обеспечению безопасности

Джерард Хольцман из НАСА предложил в статье [17] свои 10 правил разработки программ с особыми требованиями к обеспечению безопасности, которые тоже очень хорошо перекликаются с принципом KISS.

Правило 1. Ограничьте весь код очень простыми конструкциями управления – не используйте goto, setjmp или longjmp, или прямую или косвенную рекурсию. Объяснение: более простая управляющая логика транслируется на более мощные способности к анализу и часто приводит к улучшенной ясности кода.

Изгнание рекурсии – здесь, возможно, самая большая неожиданность.

Уход от рекурсии приводит к наличию нециклического графа запроса функции, который может использовать анализатор кода, чтобы оттестировать ограничения на использование стека и ограниченности исполнения. Заметьте, что это правило не требует, чтобы все функции имели единственную точку возврата, хотя это также часто упрощает управляющую логику. Тем не менее, в некоторых случаях ранний возврат ошибки является более простым решением.

Правило 2. Дайте всем циклам фиксированное верхнее предельное значение. Для инструментального средства должно быть тривиально возможным проверить предельное значение статически, что цикл не может превысить заданное верхнее предельное число повторений. Если инструментальное средство не может проверить предельное значение цикла статически, правило считается нарушенным. Объяснение: отсутствие рекурсии и присутствие предельных значений цикла предотвращает выход кода из-под контроля. Это правило, конечно, не применяется к повторениям, которые предназначаются для того, чтобы не закончиться процессу, например, в планировщике процесса. В этих специальных случаях применяется обратное правило: должно быть возможным для инструментального средства проверки проверить статически, что повторение не может закончиться. Один способ выполнять это правило состоит в том, чтобы добавить явное верхнее предельное значение ко всем циклам, которые имеют переменное число повторений, например, код, который проходит связанный список. Когда цикл превышает верхнее предельное значение, он должен запустить утверждение неудачи, и функция, содержащая неудачную итерацию, должна возвратить ошибку.

Правило 3. Не используйте динамическое распределение памяти после инициализации. Объяснение: Это правило появляется в большинстве руководящих принципов программирования для программного обеспечения с особыми требованиями обеспечения безопасности. Причина проста: распределители памяти, типа malloc, и сборщики мусора часто имеют непредсказуемое поведение, которое может значительно влиять на производительность.

Известный класс ошибок кодирования также происходит от нерационального распределения памяти и процедур освобождения: забывание освобождать память или продолжение использования памяти после того, как она была освобождена, попытки разместить больше памяти, чем физически доступно, переход границы размещенной памяти и так далее. Принуждение всех приложений жить в пределах фиксированной, предварительно размещенной области памяти может устранить многие из этих проблем и облегчить проверку использования памяти. Заметьте, что единственный путь динамически требовать память в отсутствии распределения памяти из «кучи» состоит в том, чтобы использовать память стека. В отсутствии рекурсии верхнее граничное значение на использование памяти стека может быть получено

статически, таким образом, позволяя проверить, что приложение будет всегда жить в пределах границ его ресурса.

Правило 4. Ни одна функция не должна быть длиннее того, что может быть напечатано на отдельном листе бумаги стандартного формата с одной строкой на оператор и одной строкой на декларацию. Как правило, это означает приблизительно не более 60 строк кода в функции. Объяснение: Каждая функция должна быть логической единицей в коде, который является понятным и поддающимися проверке как единица. Намного тяжелее понять логическую единицу, которая охватывает множество страниц. Чрезмерно длинные функции – часто признак плохо структурированного кода.

Правило 5. Плотность утверждений кода должна составлять в среднем минимально два утверждения на функцию. Утверждения должны использоваться для проверки аномальных состояний, которые никогда не должны случаться при исполнении в реальной жизни. Утверждения должны быть независимыми от побочных эффектов и должны быть определены как булевы тесты. Когда утверждение терпит неудачу, должно быть предпринято явное действие восстановления, типа возвращения состояния ошибки, вызывающего функцию, которая выполняет утверждение неудачи. Любое утверждение, для которого статическое инструментальное средство проверки может доказать, что оно никогда не может потерпеть неудачу или никогда не выполняется, нарушает это правило. Объяснение: статистика для промышленного объема работ по программированию указывает, что проверки блока часто находят, по крайней мере, один дефект на 10 – 100 строк письменного кода. Шансы перехвата дефектов значительно увеличиваются с увеличивающейся плотностью утверждений. Использование утверждений часто рекомендуется как часть мощной защитной стратегии программирования. Разработчики могут использовать утверждения, чтобы проверить предварительные и выходные условия функций, значения параметра, возвращаемые значений функций и переменные цикла. Поскольку предложенные утверждения независимы от побочных эффектов, они могут быть выборочно блокированы после испытания в программе с особыми требованиями по производительности.

Правило 6. Объявите все информационные объекты при наименьшем возможном уровне области действия. Объяснение: это правило поддерживает основной принцип сокрытия данных. Ясно, что, если объект не находится в области действия, другие модули не могут обратиться или разрушить его значения. Точно так же, если тестер должен диагностировать ошибочное значение объекта, чем меньше число утверждений, где значение могло бы быть назначено, тем легче диагностировать проблему. Правило также препятствует повторному использованию переменных в многочисленных, несовместимых целях, которые могут усложнить обнаружение ошибок.

Правило 7. Каждая вызываемая функция должна проверять возвращаемые

значения непустых (nonvoid) функций, и каждая вызванная функция должна проверять достоверность всех параметров, подготовленных вызывающей программой. Это, возможно, наиболее часто нарушаемое правило. Оно означает, что даже возвращаемое значение операторов printf и операторов close файла должны быть проверены. Однако, если ответ на ошибку не отличим от ответа на успех, есть определенный момент в явной проверке возвращаемого значения. Он имеет место при вызовах printf и close. В случаях, подобных этим, недвусмысленно явное или неявное преобразование типа возвращаемого значения функции к void может быть приемлемым, таким образом, указывая, что программист явно и не случайно решил игнорировать возвращаемое значение.

В более сомнительных случаях, должен быть предложен комментарий, чтобы объяснить, почему возвращаемое значение можно считать несущественным. Тем не менее, в большинстве случаев, возвращаемое значение функции не должно игнорироваться, особенно, если функция должна распространить ошибочное возвращаемое значение по цепи вызова функции.

Правило 8. Использование препроцессора должно быть ограничено включением заголовочных файлов и простых макроопределений. Не разрешаются приклеивание маркеров (token pasting), списки переменных аргументов (эллипсы) и рекурсивные макровызовы. Все макросы должны распространяться на полные синтаксические единицы. Использование директив условной трансляции должно быть сведено к минимуму. Препроцессор С – мощное инструментальное средство, которое может разрушить ясность кода и многие программы контроля на основе текста. Эффект конструкций в неограниченном коде препроцессора чрезвычайно трудно расшифровать даже с использованием формального языка определения. В новой реализации препроцессора С разработчики часто должны обращаться к использованию более ранних реализаций, чтобы интерпретировать сложный язык определения в стандарте С. Объяснение для предостережения против условной трансляции очень важно. Только с 10 директивами условной трансляции могут быть до 210 возможных версий кода, каждая из которых должна быть проверена. Использования условной трансляции можно избежать не всегда, но даже в усилиях по разработке больших программ редко есть оправдание больше, чем на одну или две таких директивы вне библиотеки стандартных текстов, что позволяет избежать многократных включений одного и того же заголовочного файла. Программы контроля на основе инструментальных средств должны помечать каждое использование, каждое из которого должно быть оправдано в коде.

Правило 9. Использование указателей должно быть ограничено. В особенности, должно использоваться не более одного уровня разыменования. Указатель операции разыменования не может быть скрыт в макроопределениях или внутри декларации typedef. Не разрешаются указатели функции. Указатели

часто используются неправильно даже опытными программистами. Они могут сделать трудным отслеживание или анализ потока данных в программе, особенно анализаторами на основе инструментальных средств. Указатели функции должны применяться, если есть веские причины для этого, потому что они могут серьезно ограничить типы автоматизированных проверок, которые могут выполнить программы контроля кода. Например, если используются указатели функции, то часто невозможно для инструментального средства доказать отсутствие рекурсии. Требуются альтернативные гарантии для компенсации этой потери в мощности проверки.

Правило 10. Код должен быть откомпилирован в первый день разработки со всеми предупреждениями компилятора, разрешенными при самой придирчивой доступной установке. Весь код должен компилироваться без предупреждений. Весь код должен также проверяться ежедневно, по крайней мере, с одним, но предпочтительно больше, чем с одним мощным анализатором статического исходного текста и должен передать все анализы с нулевыми предупреждениями. На рынке сегодня есть несколько чрезвычайно эффективных анализаторов статических исходных текстов, а также довольно много свободно распространяемых инструментальных средств. Нет оправдания не использовать эту готовую доступную технологию для разработки программного обеспечения. Это обычная практика для разработки программ без особых требований к обеспечению безопасности.

Подход «Чем хуже, тем лучше»

«Чем хуже, тем лучше» – подход к разработке программного обеспечения, объявляющий простоту реализации и простоту интерфейса более важными, чем любые другие свойства системы. Этот стиль описан Ричардом П. Гэбриелом (Richard P. Gabriel) в работе «Lisp: «Good News, Bad News, How to Win Big» в разделе «The Rise of 'Worse is Better» и часто перепечатывается отдельной статьей.

Гэбриел описывает подход следующим образом.

1. Простота: реализация и интерфейс должны быть простыми. Простота реализации даже несколько важнее простоты интерфейса. Простота – самое важное требование при выборе дизайна.
2. Правильность: дизайн должен быть правильным во всех видимых проявлениях. Простой дизайн немного лучше, чем правильный.
3. Логичность (последовательность): дизайн не должен быть слишком нелогичным. Иногда можно пожертвовать логичностью ради простоты, но лучше отказаться от тех частей дизайна, которые полезны лишь в редких обстоятельствах, чем усложнить реализацию или пожертвовать логичностью.
4. Полнота: дизайн должен охватывать как можно больше важных ситуаций.

Полнотой можно жертвовать в пользу остальных качеств и обязательно нужно жертвовать, если она мешает простоте. Логичностью можно жертвовать в пользу полноты, если сохраняется простота; особенно бесполезна логичность интерфейса.

Гэбриел считает язык C и систему Unix примерами такого подхода. В статье ему противопоставляется подход, который называется «подход MIT» (MIT – Massachusetts Institute of Technology). Гэбриел так описывает этот подход к дизайну.

1. Простота: реализация и интерфейс должны быть простыми. Простота интерфейса важнее простоты реализации.
2. Правильность: дизайн должен быть правильным во всех отношениях. Неправильный дизайн категорически запрещён.
3. Логичность так же важна, как и правильность. Ради логичности можно жертвовать простотой и полнотой.
4. Полнота: дизайн должен охватывать как можно больше важных ситуаций. Все вероятные ситуации должны быть предусмотрены. Простота не должна слишком мешать полноте.

Гэбриел утверждает, что подход «чем хуже, тем лучше» предпочтительнее «подхода MIT». Простая в реализации система будет легко перенесена под разные операционные системы, то есть быстро распространится ещё до того, как система, сделанная по принципам MIT, будет написана. Более простая в реализации система привлечёт больше пользователей, понимающих, как она работает и желающих её улучшить. Улучшения будут продолжаться, пока система не станет почти идеальной. В качестве примера Гэбриел приводит компиляторы для языков C и Лисп. В 1987 году, пишет Гэбриел, компиляторы с этих языков были почти одинаковы по качеству, но было гораздо больше желающих улучшить компилятор C, чем компилятор Лиспа. Хотя Гэбриел, возможно, первым сформулировал этот принцип, похожие идеи использовались гораздо раньше в идеологии UNIX и программного обеспечения с открытым кодом.

Сложность программного обеспечения

Ниже приведен фрагмент статьи Никалауса Вирта «Долой жирные программы»[6].

Стало правилом: всякий раз, когда выпускается новая версия программного продукта, существенно, порой на много мегабайт, подскакивают его требования к размерам компьютерной памяти. Когда такие запросы превышают имеющуюся в наличии память, приходится закупать дополнительную. Когда же дальнейшее расширение невозможно, то надо приобретать новый, более мощный компьютер или рабочую станцию. Но идут

ли большая производительность и расширенная функциональность в ногу со все увеличивающимися запросами на вычислительные ресурсы? В большинстве случаев – нет.

Лет 25 тому назад, интерактивный текстовый редактор мог быть спроектирован из расчета всего лишь 8000 байтов памяти. Современные редакторы текстов программ требуют в 100 и более раз больше. Операционная система должна была обслуживать 8000 байтов, а компилятор – умещаться в 32 Кбайт, в то время как их сегодняшние потомки требуют для своей работы многих мегабайтов. И что же, это раздутое программное обеспечение стало быстрее и эффективнее? Наоборот. Если бы не аппаратура с ее возросшей в тысячи раз производительностью, современные программные средства было бы просто невозможно использовать.

Считается, что расширенная функциональность и удобства пользователя оправдывают все возрастающие размеры программ; однако более пристальный взгляд обнаруживает шаткость подобных оправданий. Тот же текстовый редактор все еще выполняет достаточно простую задачу по вставке, удалению и переносу фрагментов текста; компилятор по-прежнему транслирует текст в исполняемый код и операционная система, как и в былые времена, управляет памятью, дисковым пространством и циклами процессора. Эти базовые обязанности вовсе не изменились с появлением окон, стратегии «вырезание-вставка» и всплывающих меню, так же как и с заменой текстовых командных строк изящными пиктограммами.

Столь явный взрывной рост размеров программного обеспечения был бы, конечно, неприемлем, если бы не ошеломляющий прогресс полупроводниковой технологии, которая улучшила соотношение цена/производительность в степени, не имеющей аналогов ни в какой другой области технологии. Так, с 1978 по 1993 год для семейства процессоров Intel 80x86 производительность увеличилась в 335 раз, плотность размещения транзисторов – в 107 раз, в то время как цена только в 3 раза. Перспективы для постоянного увеличения производительности сохраняются, при том, что нет никаких признаков, что волчий аппетит, присущий ныне программному обеспечению, будет в обозримом будущем утолен. Такое развитие событий спровоцировало появление многочисленных правил, законов и выводов, которые выражаются в универсальных терминах; как таковые, их невозможно ни доказать, ни опровергнуть. Следующие два «закона» чрезвычайно хорошо (хотя и с долей иронии) отражают нынешнее положение дел.

- Программное обеспечение увеличивается в размерах до тех пор, пока не заполнит всю доступную на данный момент память (Паркинсон).
- Программное обеспечение замедляется более быстро, чем аппаратура становится быстрее (Рейзер).

Неконтролируемый рост размеров программ принимается как должное

также и потому, что потребители имеют затруднения в отделении действительно существенных особенностей программного продукта от особенностей, которые просто «хорошо бы иметь». Примеры: произвольно перекрывающиеся окна, предлагаемые некритично воспринятой популярной метафорой «рабочий стол»; причудливые пиктограммы, декорирующие экран – антикварные почтовые ящики для электронной почты или корзинки для сбора мусора (которые, к тому же еще, остаются видимыми при движении к их новому местоположению). Эти детали привлекательны, но не существенны, и они имеют свою скрытую стоимость.

Причины громоздкости программного обеспечения Вирт формулирует следующим образом. Приводим еще один фрагмент статьи [2].

Итак, два фактора вносят вклад в приятие потребителями программного обеспечения все более растущих размеров: быстро увеличивающаяся аппаратная производительность и игнорирование принципиальной разницы между жизненно важными возможностями и теми, которые "хорошо бы иметь". Но, быть может, более важно не просто найти причины для такой толерантности, попытаться понять, что же обуславливает дрейф программного обеспечения навстречу сложности. Основной причиной является некритичное принятие фирмами-поставщиками практически любого требования потребителей относительно новых возможностей программного продукта. Всякая несовместимость с первоначальными концепциями системы либо игнорируется, либо проходит нераспознанной. В результате, проектные решения усложняются, а в использовании продукт становится более громоздким. Когда мощность системы измеряется числом ее возможностей, количество становится более важным, чем качество. Считается, что каждая новая редакция продукта должна предлагать какие-нибудь дополнительные возможности, даже если некоторые из них реально не добавляют функциональности.

Другая важная причина, ответственная за программную сложность, лежит в «монолитном» дизайне, когда все мыслимые возможности сразу закладываются в систему. Каждый потребитель платит за все возможности, но реально использует лишь немногие из них. В идеале же, должна предлагаться только базовая система с заложенными в нее существенными возможностями, но эта система должна иметь потенциал для различных расширений. Тогда каждый потребитель мог бы выбирать функции, действительно необходимые для его задачи. Возросшая производительность аппаратуры, несомненно, явилась стимулом для разработчиков при атаке на более сложные проблемы, а последние неизбежно требуют более сложных решений. Однако речь идет не о этой внутренне присущей программным системам сложности. Мы говорим здесь о сложности, искусственно привнесенной. Существует масса проблем, давно уже решенных, но ныне нам предлагаются "новые" решения тех же проблем, завернутые в более громоздкую программную оболочку.

Возросшая сложность по большей части является следствием наших недавно возникших пристрастий к «дружественному» пользовательскому интерфейсу. Я уже упоминал окна и пиктограммы; сюда же можно добавить цвет, полутона, тени, всплывающие меню, всевозможные картинки и диалоговые «реквизиты» различных типов.

Unix как пример красивой системы

Линус Торвальдс в своей книге «Just for fun»[4] рассматривает Unix как красивую систему, олицетворяющую некую систему ценностей. Приведем фрагмент книги.

Unix характерна тем, что она утверждает некоторые базовые ценности. Это цельная и красивая операционная система. Она избегает особых случаев. В Unix есть понятие процесса: процесс – это все, что что-нибудь делает. Простой пример. В Unix команда оболочки, которую вводят, чтобы войти в систему, не встроена в операционку, как в DOS. Это просто задание. Ничем не отличающееся от остальных. Просто это задание читает с клавиатуры и пишет на монитор. В Unix все, что что-то делает – процесс. А еще там есть файлы.

Простота структуры Unix всегда поражала меня, как и большинство людей по крайней мере, нас, хакеров). Почти все, что делается в Unix, выполняется с помощью шести базовых операций (называемых "системными вызовами", потому что они представляют из себя вызовы системы для выполнения тех или иных действий). Из этих шести базовых вызовов можно построить почти все на свете.

Одной из фундаментальных операций Unix является «операция порождения (fork)». Выполняя «fork», процесс создает свою точную копию. Таким образом вы получаете две идентичные копии. Порожденная копия чаще всего выполняет другой процесс – заменяет себя новой программой. Это вторая базовая операция. Оставшиеся четыре вызова – open (открыть), close (закрыть), read (читать) и write (писать) – предназначены для доступа к файлам. Эти шесть системных вызовов представляют собой простые операции, из которых и состоит Unix.

Конечно, есть еще куча других системных вызовов, которые осуществляют детализацию. Но если вы поняли шесть базовых – вы поняли Unix. Потому что одна из прелестей Unix в том, что для создания сложных вещей не нужны сложные интерфейсы. Любого уровня сложности можно достичь за счет сочетания простых вещей. Для решения сложной проблемы нужно лишь создать связи («каналы» в терминологии Unix) между простыми процессами.

Плохо, когда для любого действия у системы есть специальный интерфейс. В Unix все наоборот. Она предоставляет строительные блоки, из которых можно создать что угодно. Вот что такое стройная архитектура.

То же самое с языками. В английском 26 букв, и с их помощью можно написать все. А в китайском для каждой мыслимой вещи своя буква. В китайском вы сразу же получаете в свое распоряжение сложные вещи, которые можно комбинировать ограниченным образом. Это больше напоминает подход VMS: есть множество сложных вещей с интересным смыслом, которые можно использовать только одним способом. И в Windows то же самое.

В Unix, напротив, основная идея: «чем меньше, тем красивее». Здесь есть небольшой набор простых базовых строительных блоков, из которых можно строить бесконечно сложные конструкции.

Именно так, кстати, обстоит дело и в физике. Эксперименты позволяют открыть фундаментальные законы, которые, как предполагается, крайне просты. Сложность мира возникает за счет множества удивительных взаимосвязей, которые можно вывести из этих простых законов, а не из внутренней сложности самих законов.

Простота Unix не возникла сама по себе. Unix со своей концепцией простых строительных блоков была кропотливо разработана Деннисом Ричи и Кеном Томпсоном в Bell Labs компании AT&T. Простоту вовсе не следует отождествлять с легкостью. Простота требует проектирования и хорошего вкуса.

Если вернуться к примеру с языками, то пиктографическое письмо – например, египетские или китайские иероглифы – кажется примитивнее, а подход, использующий строительные блоки, требует гораздо более абстрактного мышления. Точно так же и простоту Unix не следует путать с отсутствием изощренности.

Из этого вовсе не следует, что создание Unix было вызвано какими-то сложными причинами. Как часто бывает в компьютерной области, все началось с игр. Нужно было, чтобы кто-то захотел играть в компьютерные игры на PDP-11. Именно из этого выросла Unix – персонального проекта Денниса и Кена, пожелавших играть в «Звездные войны». А поскольку этот проект никто не воспринимал всерьез, AT&T не занималась коммерческим применением Unix. AT&T была регулируемой монополией и все равно не могла, например, продавать компьютеры. Поэтому создатели Unix стали бесплатно предоставлять ее вместе с лицензиями на исходные тексты всем желающим, в особенности университетам. Они относились к этому просто[4].

3. Аппаратно-зависимое ПО ВВС

3.1. Особенности реализации

Ни для кого не секрет, что аппаратно-зависимое программное обеспечение создавать, как правило, значительно сложнее, чем писать обычные прикладные программы. Классические программисты-прикладники как огня боятся «железа», а если и берутся его программировать, то у них очень часто получаются довольно плачевные результаты. Почему так происходит?

Первая причина в том, что аппаратное обеспечение работает совсем не в той модели вычислений, к которой привыкли прикладные программисты. К сожалению, система подготовки такова, что большинство программистов никогда не выходило за пределы модели вычислений на базе машины Фон-Неймана. Работа с аппаратным обеспечением требует понимание таких вещей как параллелизм, концепцию прерываний, понимание комбинационных схем и конечных автоматов не в теории, а на практике.

Вторая причина кроется в том, что для написания грамотного программного обеспечения необходимо понимать схемотехнику аппаратной части проектируемой системы. Уровень понимания аппаратуры должен быть таким, чтобы программист при необходимости сам мог составить схему или понять и откорректировать схему, предложенную ему аппаратчиком. К счастью, процесс изучения аппаратной части можно сильно упростить, заменив длительную разработку и прототипирование реальных устройств на работу с моделями. Сейчас появились мощные современные средства кодизайна и косимуляции, такие, например, как Multisim или Proteus.

Третья причина состоит в том, что аппаратно-зависимое программное обеспечение тесно связано с решаемой прикладной задачей. Программировать аппаратуру «вообще», в отрыве от реальной задачи невозможно. Для написания полноценного системного программного обеспечения необходимо создавать тестовые стенды и имитаторы окружения. В создании таких средств нам могут помочь пакет моделирования MATLAB и виртуальная лаборатория LABView.

Четвертая проблема состоит в сравнительно малых вычислительных ресурсах. Этот факт заставляет отказаться от большинства привычных обычным программистам инструментальных средств.

К сказанному выше необходимо добавить, что системное программное обеспечение является платформой для других программ. Этот факт заставляет серьезнее думать о реализации различных механизмов надежности и безопасности.

3.2. Уровень абстракции от аппаратуры (HAL)

Уровень абстракции от аппаратуры, HAL (Hardware Abstraction Layer) – программно-реализованная платформа, скрывающая особенности реализации аппаратных средств, сделанных по-разному или выпущенных разными производителями. Цена такой унификации – дополнительные накладные расходы на реализацию дополнительного уровня. Как правило, HAL имеет смысл вводить на сравнительно мощных микроконтроллерах, если у вас стоит задача создать легко переносимую систему.

HAL позволяет унифицировать все интерфейсы всех аппаратных устройств системы. В простых системах такую унификацию выполняют на уровне драйверов.

HAL имеет смысл разрабатывать в следующих случаях.

- Аппаратное обеспечение часто модернизируется. Такие модернизации очень вероятны при мелкосерийном производстве.
- Необходимо выпустить программное обеспечение для целого ряда аппаратных платформ.
- Драйверы устройств сложные и их перенос требует значительных ресурсов.

Недостатком HAL является снижение производительности системы из-за введения дополнительного уровня. В некоторых случаях, например, когда аппаратные платформы кардинально отличаются друг от друга, использование HAL нецелесообразно.

Реализация HAL на примере ОС РВ eCos

В ОС РВ eCos HAL разбит на три уровня. Первый уровень называется архитектурный HAL (architecture HAL). Он скрывает основные особенности центрального процессора, системы прерываний и способа переключения контекста. Второй уровень называется уровень различий (variant HAL). Этот уровень предназначен для описания таких особенностей, как кеш центрального процессора, математический сопроцессор, контроллеры прерываний, память и т.п. Третий уровень называется платформный HAL (platform HAL). На этом уровне описывается старт системы (startup), регистры системы ввода вывода, таймеры, контроллер прерываний [14,26].

Описанные уровни HAL в таком виде существуют только в справочном руководстве по eCos. На самом деле, из-за особенностей реализации различных процессоров и микроконтроллеров в реальных исходных текстах ОС РВ присутствуют не все уровни, кроме того, границы между уровнями сильно размыты.

Обзор исходных текстов eCos-2.0, взятых с официального сайта проекта (http://http://ecos.sourceware.org/), показал следующий ряд фактов. Большая часть HAL реализована на языках Си и ассемблера. Интерфейсы реализованы в виде макросов. Уровни в HAL действительно не имеют четких границ, видимо, это связано с сильными отличиями различных платформ друг от друга. Итак, что реализовано в HAL?

1. Инициализация различных подсистем после включения питания, в том числе процессора, математического сопроцессора, контроллеров, монитора HAL, системы прерываний.
2. Макросы для управления кэшем процессора.
3. Макросы для работы с прерываниями: разрешение, запрещение, подключение и отключение обработчиков, управление контроллером приоритетных прерываний.
4. Макросы для работы с часами: инициализация, чтение времени. В данном случае речь идет о системных часах.
5. Макросы для реализации доступа к 8, 16 и 32 разрядным регистрам с различным порядкам байтов (младший байт вперед, старший байт вперед).

Рассмотрим для примера несколько простых макросов, реализованных в HAL eCos-2.0, чтобы вы могли иметь о нем минимальное представление. Для сравнения проанализируем реализацию части HAL для платформ Intel i386 и ARM. Макросы для разрешения и запрещения прерываний для платформы i386 находятся в файле hal_inr.h и имеют примерно следующий вид (к моменту прочтения этой книги их могут изменить):

```
// CPU interrupt enable/disable macros

#define HAL_ENABLE_INTERRUPTS()                 \
CYG_MACRO_START                                 \
    asm ("sti") ;                               \
CYG_MACRO_END

#define HAL_DISABLE_INTERRUPTS(_old_)           \
CYG_MACRO_START                                 \
    register int x ;                            \
    asm volatile (                              \
        "pushfl ;"                              \
        "popl %0 ;"                             \
        "cli"                                   \
        :"=r" (x)                          \
        ) ;                                     \
    (_old_) = (x & 0x200);                      \
CYG_MACRO_END
```

Макросы для разрешения и запрещения прерываний для платформы ARM также находятся в файле hal_inr.h. Наполнение уже другое, система прерываний процессора с ядром ARM довольно сильно отличается от Intel x86.

```
// Interrupt control macros

#ifndef __thumb__
// Note: This disables both FIQ and IRQ interrupts!
#define HAL_DISABLE_INTERRUPTS(_old_)           \
    asm volatile (                              \
        "mrs %0,cpsr;"                          \
        "mrs r4,cpsr;"                          \
        "orr r4,r4,#0xC0;"                      \
        "msr cpsr,r4"                           \
        : "=r"(_old_)                           \
        :                                       \
        : "r4"                                  \
        );

#define HAL_ENABLE_INTERRUPTS()                 \
    asm volatile (                              \
        "mrs r3,cpsr;"                          \
        "bic r3,r3,#0xC0;"                      \
        "msr cpsr,r3"                           \
        :                                       \
        :                                       \
        : "r3"                                  \
        );
```

Из примеров видно, что макросы HAL_DISABLE_INTERRUPTS и HAL_ENABLE_INTERRUPTS реализованы по-разному.

3.3. Драйверы устройств ВВС

Драйвер – преобразователь информации, позволяющий соединить уровень клиента и уровень обслуживаемого устройства. Драйвер является интерфейсом между клиентом и устройством.

В рамках терминологии систем «клиент-сервер» драйвер является

сервером, обслуживающим некоторое устройство.

В аппаратном обеспечении эквивалентом драйвера является контроллер.

В задачи драйвера входит:

- инкапсуляция свойств устройства;
- упрощение и стандартизация интерфейсов сходных устройств;
- первичная обработка данных, поступающих из/в устройство;
- диспетчеризация доступа к устройству.

Архитектура драйвера должна соответствовать архитектуре (идеологии) всей системы (например, ОС РВ).

Примеры драйверов различных устройств

Пример драйвера последовательного канала ОС РВ eCos

В операционных системах реального времени на первое место выдвигаются требования переносимости. Как правило, драйверы, написанные под такие платформы, достаточно громоздкие. Рассмотрим пример функции для помещения символа в последовательный канал из исходных текстов драйверов операционной системы eCos-2.0 для платформы ПК (i386).

```
// Передача символа в выходной буфер устройства
// Возвращает 'true' если символ отправлен в устройство
bool pc_serial_putc(serial_channel *chan, unsigned char c)
{
  cyg_uint8 _lsr;
  pc_serial_info *ser_chan = (pc_serial_info *)chan->dev_priv;
  cyg_addrword_t base = ser_chan->base;

  HAL_READ_UINT8(base+REG_lsr, _lsr);
  if (_lsr & LSR_THE) {
      // Transmit buffer is empty
      HAL_WRITE_UINT8(base+REG_thr, c);
      return true;
  }
  // No space
  return false;
}
```

Из примера видно, что драйвер использует макросы HAL более низкого уровня. Для платформы PC (i386) драйвер последовательного канала включен в HAL. Для большинства других платформ исходные тексты драйвера последовательного канала можно посмотреть в каталоге ecos-2.0/packages/io/serial/src/common/.

Пример драйвера последовательного канала без ОС РВ

Для контроллеров, не имеющих ОС РВ, драйверы пишутся несколько иначе. На первое место обычно выдвигается эффективность реализации. Драйверы не являются универсальными (в отличии от реализации для ОС РВ), то есть они не подходят на все случаи жизни. Как правило, драйверы реализуют для решения какого-либо сравнительно узкого класса задач. Совместимость с такими драйверами может быть достигнута лишь на уровне интерфейсов, да и то, не во всех случаях.

Самый простой способ реализации драйвера – работа с UART по опросу. Ниже приведен пример такого драйвера для Philips LPC2000.

```
// Пример драйвера UART, работающего по опросу

// Инициализация
static void init_serial0( void )
{
    U0LCR   = 0x83;
    U0DLL   = 78;          // X = Fosc / (16 * 9600)
    U0DLM   = 0x00;
    U0LCR   = 0x03;        // DLAB = 0
}

// Передача байта
void wsio0 (unsigned char ch)
{
    reset_wdt();
    while ( ! ( U0LSR & 0x20 ) );
    U0THR = ch;
}

// Прием байта
unsigned char rsio0 (void)
{
    while ( ! ( U0LSR & 0x01 ) )       reset_wdt();
    return ( U0RBR );
}
```

Достоинство такого драйвера – крайняя простота реализации. Пожалуй, это его единственный плюс. В остальном этот драйвер имеет множество недостатков. Во-первых, передача каждого очередного байта блокирует процессор на все время передачи этого байта. Например, при скорости обмена 9600 бит в секунду процессор будет работать в холостую целую миллисекунду. Во-вторых, на интерпретацию принятого байта у приемной стороны есть фиксированное время. Если не успеть обработать принятый байт за время передачи очередного байта, информация будет потеряна. Применять такие драйвера имеет смысл при тестировании в очень простых и учебных проектах.

Как правило, в большинстве случаев драйвер последовательного канала (см. пример для процессора Fijitsu) реализуется с использованием прерываний.

Реализуется две очереди FIFO. Одна очередь используется для приема, другая для передачи. Очереди нужны для выравнивания скоростей прикладных процессов (тех которые формируют данные для передачи и обрабатывают принятые данные) и процессами приема и передачи данных через последовательный канал. Когда очередь заполняется, происходит потеря данных для приемной стороны или блокировка процесса для передающей.

```
/*-------------------------------------------------------------
Функция передачи байта данных в последовательный канал для Fujitsu BM90F590.
Если в передаваемом буфере нет свободного места, то блокировки не происходит.
data - передаваемый байт
результат OK, если данные положены в передаваемый буфер
------------------------------------------------------------- */
ERRCODE WriteSerial( BYTE data )
{
    //если исходящий буфер переполнен, возвращаем ошибку
    if ( tx_count>=TX_FIFO_SIZE ) return ERR_SERIAL_TXFULL;

    USR0 &= ~0x04;
    tx_buf[(tx_start+tx_count)&(TX_FIFO_SIZE-1)] = data;
    ++tx_count;
    USR0 |= 0x04;
    return OK;
}
/*-------------------------------------------------------------
Обработчик прерываний по передаче байта по последовательному каналу.
------------------------------------------------------------- */
__interrupt void Serial0TxHandler(void)
{
    if ( USR0&0x10 ) {
        if ( tx_count ) {
            UODR0 = tx_buf[tx_start&(TX_FIFO_SIZE-1)];
            --tx_count;
            ++tx_start; } else
        USR0 &= ~0x04;
    }
}
```

Драйвер такого типа, как правило, состоит из пяти основных частей.

1. Функция инициализации.
2. Функция занесения данных в выходной буфер FIFO.
3. Функция чтения данных из входного буфера FIFO.
4. Обработчик прерывания по приему данных (кладет данные во входной буфер FIFO).
5. Обработчик прерывания по передаче данных (берет данные из выходного буфера FIFO).

С прерываниями по приему все просто. Приняли байт, произошло прерывание, обработчик поместил принятый байт в FIFO. При передаче все несколько сложнее. Обычно для того, чтобы произошло прерывание для

передачи, микроконтроллеров достаточно.

Для исключения возможности блокировки и потери данных в драйвер последовательного канала можно добавить квитирование. Аппаратное квитирование (сигналы CTS, RTS) требует дополнительных линий связи. Программное квитирование не требует наличия дополнительных линий связи, но вам придется передавать информацию только в ASCII форме, так как для квитирования используются символы Xon и Xoff. И тот и другой способ квитирования имеют свои плюсы и минусы. Оба способа квитирования могут серьезно усложнить реализацию драйвера.

Драйвер часов реального времени

Для встраиваемых систем, работающих с астрономическим временем, часы реального времени – один из важнейших узлов системы. Пример такой системы – контроллер управления городским освещением. Дело в том, что в городах и на крупных заводах освещение включают и выключают по расписанию. Отклонение от расписания чревато перерасходом электроэнергии в случае более раннего включения (в масштабах города это очень много) или судебными исками пострадавших, в том случае, если стало темно, а освещение еще не включено.

```
    /* Драйвер RTC M4156 */

   /* Проверка корректности времени и даты */
static char check_td( const TimeDate * td )
{
    if(   td->seconds > 59 )     return ERROR;
    if(   td->minutes > 59 )     return ERROR;
    if(   td->hours   > 23 )     return ERROR;
    if( ( td->day     > 7  ) ||
        ( td->day    == 0  ) )   return ERROR;
    if( (  td->date   > 31 ) ||
        (  td->date  == 0  ) )   return ERROR;
    if( (  td->month   > 12 ) ||
        (  td->month   == 0 ) )  return ERROR;
    if( (  td->years   > 20 ) ||
        (  td->years   < 04 ) )  return ERROR;
    return OK;
}

/*-------------------------------------------------------------
Чтение времени и даты
td - считанные значения времени и даты
результат:  OK    - время успешно считано, ERROR - ошибка при работе по I2C
------------------------------------------------------------- */
BYTE ReadRTC( TimeDate* td )
{
    i2cAddr = 0xD0;
    i2cTx = 1;
    i2cRx = 10;
    i2cBuffer[0] = 0;
    if (I2C_OK != i2cExchange()) {
        mmgsm_status &= ~MMGSM_STAT_RTC_ACCESS;
        return ERROR; } else
```

```
        mmgsm_status |= MMGSM_STAT_RTC_ACCESS;
    //преобразуем считанные данные из BCD формата
    td->seconds = Bcd2Int(i2cBuffer[1] & 0x7F);
    td->minutes = Bcd2Int(i2cBuffer[2] & 0x7F);
    td->hours   = Bcd2Int(i2cBuffer[3] & 0x3F);
    td->day     = Bcd2Int(i2cBuffer[4] & 0x07);
    td->date    = Bcd2Int(i2cBuffer[5] & 0x3F);
    td->month   = Bcd2Int(i2cBuffer[6] & 0x1F);
    td->years   = Bcd2Int(i2cBuffer[7]);

    if( i2cBuffer[9] != (BYTE)( ~i2cBuffer[10] ) )  {
        mmgsm_status &= ~MMGSM_STAT_RTC_TIME;
        return RTC_FAIL;  }

    if( OK != check_td( td ) ) {
        mmgsm_status &= ~MMGSM_STAT_RTC_TIME;
        return ERROR;  }
    return OK;
}
```

Какие проблемы могут встретиться вам при работе с часами? Давайте проанализируем. Итак, возможны сбои в следующих местах.

1. Питание часов (сбилось время).
2. Внутри часов, например, из-за помех (сбилось время).
3. В интерфейсе между часами и микроконтроллером (например, из-за ошибок монтажа, могут быть поставлены некорректные номиналы резисторов в шине I2C). Считываемые данные иногда корректные, иногда нет.
4. Ошибка памяти в контроллере (например, из-за помех). Данные будут искажены.
5. Ошибка в программном обеспечении. Данные будут искажены.
6. Уход частоты кварцевого резонатора из-за механического повреждения или изменения температуры.

Любая из перечисленных причин может привести к некорректной работе целой системы. Исходя из вышесказанного, при проектировании ответственной системы необходимо озаботиться созданием защитных механизмов, сводящих риск сбоев к минимуму.

В приведенном выше примере показан фрагмент драйвера RTC M4156 с простой функцией проверки корректности полученного значения времени и даты.

Драйвер аналоговых портов ввода-вывода

Оцифровка аналоговых величин и получение аналоговых величин (токов и напряжений) возможны при помощи аналого-цифровых и цифро-аналоговых преобразователей.

При работе с АЦП необходимо помнить, что процесс оцифровки происходит не мгновенно, а требует некоторого времени. Ниже приведен пример исходного текста, в котором опрос готовности АЦП производится в простом цикле.

```
//Пример простого драйвера АЦП для PIC16
unsigned short GetADC(unsigned char adcn)
{
unsigned short Res;
unsigned char cTmp,i;
    ADCON0  = 0x40 | adcn | 0x1;  //FOSC/8
    ADIF=0;
    for(i=0;i<100;i++) cTmp++;
    ADCON0      |=  0x4;          //Start A/D conversion
    while(!ADIF);
    Res=ADRESH;
    Res <<=2;
    cTmp=ADRESL;
    cTmp >>= 6;
    Res += cTmp;
    return Res;
}
```

Такое решение является не только самым простым в реализации, но, к сожалению, и самым неэффективным. Дело в том, что во время опроса готовности АЦП процессор простаивает. Такие решения используются обычно в тестах, при обучении и для очень простых проектов. Для работы с АЦП в большинстве обычных проектов обычно используют прерывания или прямой доступ к памяти. В некоторых микроконтроллерах есть целые контроллеры ввода-вывода, позволяющие получать от АЦП данные в автоматическом режиме, не задействуя ресурсы центрального процессора.

При работе с аналоговым вводом-выводом приходится решать большое количество различных проблем. Во-первых, такие системы делают очень часто многофункциональными, что заставляет задуматься о многоуровневой структуре драйвера и системе настроек. Во-вторых, любая аналоговая схема индивидуальна из-за некоторого разброса номиналов резисторов и конденсаторов. Следовательно, нужно думать об индивидуальной калибровке каждого канала. В-третьих, аналоговые цепи чувствительны к температуре. Следовательно, калибровку нужно производить для нескольких температурных диапазонов.

Нельзя забывать, что считываемые с аналоговых входов значения могут содержать помехи. Для фильтрации придется использовать какие-либо фильтры. Например, можно отбрасывать крайние значения (минимальные и максимальные), производить усреднение и медианную фильтрацию.

Рассмотрим медианную фильтрацию чуть подробнее. Медианный фильтр позволяет убрать одиночные выбросы из потока оцифрованных данных.

Потоковый медианный фильтр включается между приемником и источником сигнала. Суть его работы состоит в следующем:

- фильтр помнит три последних значения (скользящее окно);
- из этих трех значений выбирается среднее ;
- на выход фильтра попадает среднее значение.

Необходимо заметить, что под средним значением понимается не среднее арифметическое, а значение, удовлетворяющее условию (y1 >= y >= y2).

Если количество входных значений равно N, то количество выходных будет N-2.

В приведенном ниже примере фильтр рассчитан на значения от двух источников (внутренние массивы двухэлементные).

```
/*-------------------------------------------------------------
        Потоковый медианный фильтр

Вход   new_value   вход фильтра
Выход  status      1 - данные готовы
                   0 - данные не готовы (не готовыми могут быть первые два
значения
                   поданные на вход фильтра).
Результат          Отфильтрованное значение
------------------------------------------------------------- */
#include <med_filter.h>
unsigned short stream_median_filtration( unsigned short
                                    new_value,
unsigned char *status, unsigned char i )
{
static unsigned short a[ 3 ], b[ 3 ], c[ 3 ];
// скользящее окно на 3 значения для двух источников

static int             state[ 3 ] = {0, 0, 0};
// Состояние автомата медианного фильтра
char                   stat = 0;
// статус готовности данных
    // Заполняем переменные a, b, c значениями из потока
    switch( state[ i ] )
    {
    case 0: a[ i ] = new_value; stat = 0; state[ i ] = 1;
    break;
    case 1: b[ i ] = new_value; stat = 0; state[ i ] = 2;
    break;
    case 2: c[ i ] = new_value; stat = 1; state[ i ] = 3;
    break;
    case 3: a[ i ] = new_value; stat = 1; state[ i ] = 4;
    break;
    case 4: b[ i ] = new_value; stat = 1; state[ i ] = 5;
    break;
    case 5: c[ i ] = new_value; stat = 1; state[ i ] = 3;
    break;
    default:                       state[ i ] = 0; break;
    }
    // Возвращаем статус
```

```
    *status = stat;
    // Выбираем среднее значение
    if( stat )
    {
        if( a[ i ] >= b[ i ] )
        {
            if( a[ i ] >= c[ i ] )
            {
                if( b[ i ] >= c[ i ] )
                    return b[ i ];
                else
                    return c[ i ];
            }
            else
                return a[ i ];
        }
        else
        {
            if( b[ i ] >= c[ i ])
            {
                if(a[ i ] >= c[ i ])
                    return a[ i ];
                else
                    return c[ i ];
            }
            else
                return b[ i ];
        }

    }
    else
        return new_value;
}
```

Создание точного измерительного прибора на базе микроконтроллера может потребовать от вас реализации достаточно сложного драйвера.

Драйвер дискретных портов ввода-вывода

Драйвер дискретного порта ввода-вывода значительно проще аналогового, но и здесь есть свои проблемы. Во-первых, при вводе дискретной информации существует проблема дребезга контактов. Импульс, на который должен среагировать дискретный порт, может начинаться с последовательности коротких импульсов-помех, которые заставят его ложно срабатывать и загрузят центральный процессор ненужной обработкой событий. Во-вторых, в большинстве случаев требуется иметь настройки, позволяющие обеспечивать срабатывание на импульсы с определенной длительностью, полярностью, по фронту, спаду или по наличию необходимого уровня импульса.

При формировании импульсов необходимо помнить, что если мы управляем, например, каким-либо ответственным оборудованием, необходимо где-то хранить текущее состояние. Если контроллер внезапно перезапустится из-за какого-либо сбоя или помех, то у нас будет возможность быстро восстановить предыдущее значение на выходе портов.

4. Распределенные вычисления, параллелизм

Встраиваемые системы работают в реальном, а не виртуальном мире с реальными объектами. В управляющих системах могут одновременно происходить сотни событий, множество процессов взаимодействуют друг с другом в рамках одной или нескольких вычислительных систем.

К сожалению, последовательный стиль мышления для решения задач управления оказывается не приемлемым. Для того, чтобы спокойно решать задачи в области создания программного обеспечения, разработчик должен хорошо разбираться в потоковой вычислительной модели, глубоко понимать такие термины как данные, поток данных, информация и процесс.

4.1. Данные, поток данных, информация, процесс

Поток данных – последовательность данных, передаваемая от одного процесса к другому. Термин поток данных обычно применяется в рамках потоковой модели или модели вычислений сеть потоков данных.

Данные в информатике – информация, представленная в формализованном виде, что обеспечивает возможность ее хранения, обработки и передачи.

Информация (от лат. Informatio – разъяснение – изложение), первоначальная – сведения, передаваемые людьми устным, письменным или другим способом (с помощью условных сигналов, технических средств и т. д.); с середины 20 века общенаучное понятие, включающее обмен сведениями между людьми, человеком и автоматом, автоматом и автоматом; обмен сигналами в животном и растительном мире; передачу признаков от клетки к клетке, от организма к организму; одно из основных понятий кибернетики.

Процесс – вычислительная единица, предназначенная для преобразования потока информации. Процесс имеет начальную и конечную стадии своей жизни. Процесс протекает во времени. Процесс может предавать, принимать данные или управлять другими процессами. Процесс предназначен для:

- распределения ресурсов ЦП на несколько потребителей;
- изоляция (при наличии механизма защиты памяти) алгоритмов друг от друга;
- упрощение алгоритмов, обслуживающих несколько устройств (объектов) одновременно.

В широком смысле, процесс – последовательная смена состояний, явлений, ход развития чего-то.

Процессом можно назвать изменение чего-либо во времени, изменение структуры объекта или системы. Примеры процессов:

- процесс или поток операционной системы;

- работа какой-либо программы;
- проектирование;
- жизнедеятельность организма.

4.2. Потоковая модель

Потоковая модель вычислений (dataflow) представляется в виде графа, узлами которого являются процессы, а дугами – потоки данных. Потоковые модели предполагают взаимодействие двух и более процессов.

В рамках нотации потоковых диаграмм DFD (Data Flow Diagram) и CFD (Control Flow Diagram) процессы изображают кружком. В DFD процессы взаимодействуют посредством передачи данных (сплошные стрелки), а в CFD – посредством передачи управления (пунктирные стрелки). Стрелки показывают направление и способ передачи данных или управления (синхронный или асинхронный). Очень часто при проектировании встраиваемых систем смешивают DFD и CFD в рамках одного рисунка.

Диаграмма потоков управления (CFD, Control Flow Diagram) – разновидность потоковой модели, передающей в своей семантике только аспект управления процессами.

Модель управления содержит информацию о том, какие решения принимает система в ответ на изменение внешних или внутренних воздействий или режимов функционирования. Принятие решения, иначе называемого ответной реакцией системы, заключается в том, что происходит изменение состояния системы, влекущее за собой включение или выключение некоторых процессов, определенных на DFD и CFD. Помимо воздействий модель управления описывает, какую информацию о состоянии внешних сущностей необходимо получить системе и какие сигналы о собственном состоянии передать окружению [27].

Основной акцент в потоковых моделях делается на способах обмена данными между процессами. Таких способов в вычислительной технике немного. Механизмом передачи данных могут быть:

- общая память (хранилище данных). Используется в диаграммах потоков данных Йордона и в стратегии проектирования систем реального времени Пирбхая и Хатли;

- очереди (сообщения, поток данных). Используется в сетях процессов Кана.

Процесс может порождать данные в виде одного или нескольких потоков. Процесс может принимать один и более потоков данных. Поток данных может

соединять процесс с процессом, а может процесс с хранилищем данных. Обрабатывая входные данные, процесс формирует выходные.

При передаче данных через общую память возможна параллельная запись и параллельное чтение несколькими процессами. Как правило, передача данных через общую память приводит к дополнительным сложностям и путанице при проектировании.

История развития

Потоковые модели вычислений получили толчок в развитии в начале 60-х годов 20 века. На данном этапе развития вычислительной техники разработчики столкнулись с новой для себя проблемой – созданием сложных, распределенных цифровых систем. Появление насущной необходимости в одновременной обработке нескольких (двух и более) потоков данных в рамках одной вычислительной системы привело к тому, что старые подходы к синтезу и анализу систем не давали существенных результатов и проекты стали слишком сложными для понимания и реализации. В 1962 году для описания распределенных систем были придуманы сети Петри. В 1963 г. Естрином и Турном (Estrin and Turn) были предложены первые потоковые модели. Карп и Миллер (Karp and Miller) предложили в 1966 году графы без переходов. Формализация и расширение модели Эстрина была произведена Родригесом в 1969 г. Чемберлен (Chamberlain) предложил потоковый язык в 1971 г. В 1974 Каном (Kahn) были предложены сети процессов с бесконечными очередями. В этом же году Денисом (Dennis) создана потоковая модель с буферами на один элемент. Арвинд и Гостелов (Arvind and Gostelow), а также независимо Гард и Ватсон (Gurd and Watson) предложили потоковую модель с тегированными элементами. В 1975 году Йордон и Константайн (Yourdon and Constantine) предложили методику структурного проектирования программного обеспечения с декомпозицией системы на процессы и потоки данных.

Применение потоковых моделей на практике

Для примера рассмотрим потоковую модель драйвера последовательного канала, работающего по прерыванию. В этой модели есть три процесса: контроллер последовательного канала UART; обработчик прерывания; пользовательский процесс. Информация, полученная от UART, записывается процессом «обработчик прерываний» в очередь RFIFO. Пользовательский процесс может забрать байт из FIFO тогда, когда у него для этого будет возможность (т.е. очередь не будет пуста). После записи байт в WFIFO пользовательский процесс может заниматься своими делами. Обработчик прерывания заберет байт из WFIFO тогда, когда буфер передатчика UART опустеет (при этом обработчик будет вызван).

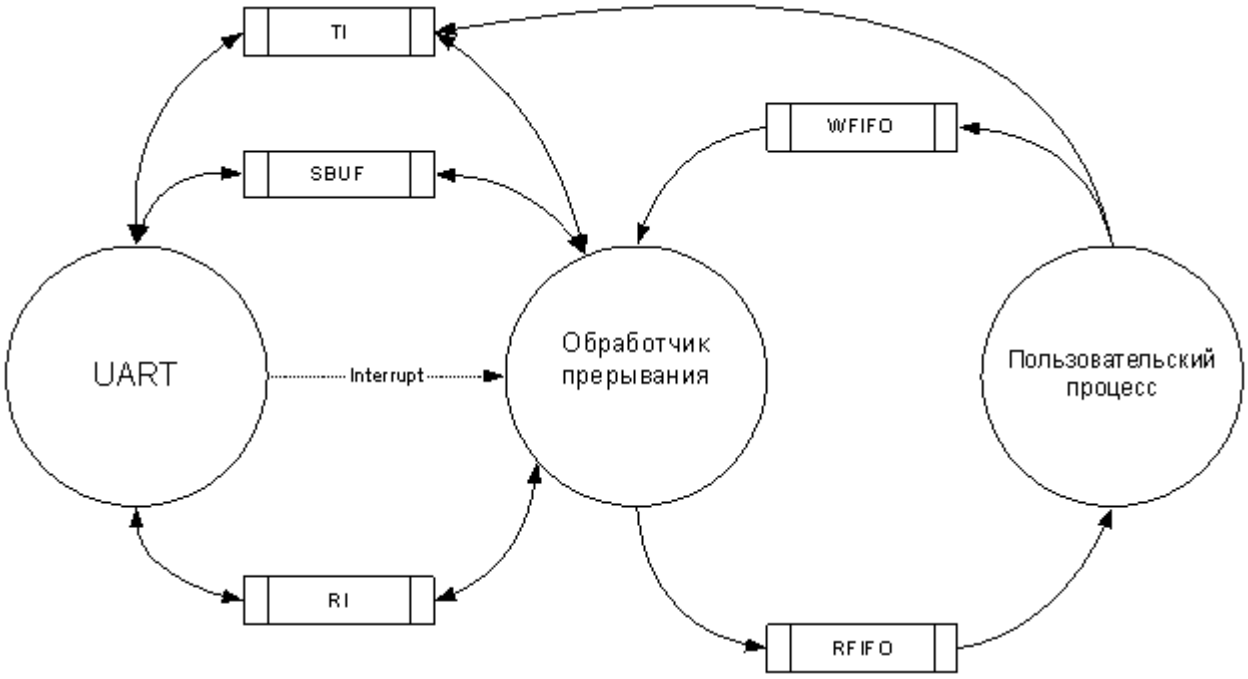


Рисунок 9. DFD модель драйвера последовательного канала

Как мы видим, не все аспекты работы потоковой модели драйвера последовательного канала для MCS51 описываются с помощью потоковой модели. Непонятно, когда происходит то или иное событие и что приводит к запуску того или иного процесса.

Проблемы реализации потоковых моделей

В классических однопроцессорных компьютерных системах, построенных на базе модели Фон-Неймана, возникает проблема, связанная с необходимостью выделения ресурса центрального процессора для процессов, т.е. так называемого планирования. Кроме этого, возникает целый ряд других проблем:

- критическая секция;
- гонки;
- взаимное исключение;
- блокировка процессов.

Модель вычислений Кана

Process Network (PN), сеть процессов, сеть потоков данных – модель вычислений, в которой система представляется в виде ориентированного графа,

вершины которого представляют собой процессы (вычисления), а дуги – упорядоченные последовательности элементов данных. Граф обычно имеет иерархичную структуру, так что вершина одного графа сама по себе является другим ориентированным графом. Вершины могут представлять собой также процедуры на некоторых языках программирования. Передача данных между вычислительными вершинами осуществляется через буферы типа FIFO, а сами вычислительные вершины постоянно (вне времени) осуществляют обработку входных данных, порождая наборы выходных данных. Модель вычислений с потоками данных (Dataow Process Networks) была предложена Каном в 1974 году для описания программных систем обработки потоков данных, отличающихся параллелизмом и независимостью отдельных ее этапов. В дальнейшем, эта модель была развита и получила распространение в области систем потоковой обработки данных (реализация аудио- и видеоприложений, обработки изображений и сигналов). Были предложены многочисленные варианты и ограничения модели, такие как Synchronous Dataflow, Structured Dataflow, Well-behaved dataflow, Boolean Dataflow, Multidimensional SDF, Integer Dataflow, Heterogeneous Dataflow и т.д., дающие те или иные дополнительные полезные свойства вычислительного процесса.

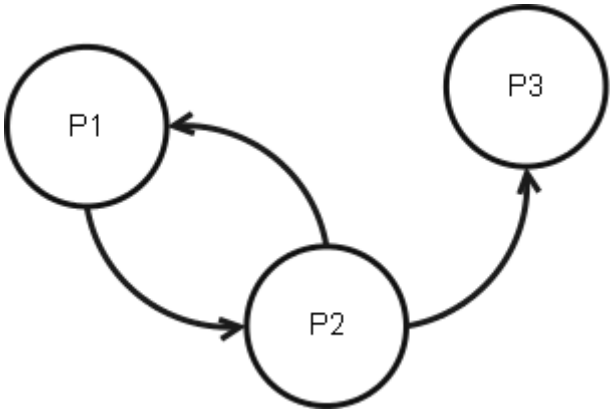


Рисунок 10. Сеть процессов. Кружками обозначены процессы, а стрелками – потоки данных

Компоненты системы (процессы) связаны каналами данных, каждый из которых представляет собой очередь с дисциплиной FIFO. Данные читаются из канала процессом в том порядке, в котором они были в него записаны. Процесс может либо читать данные из канала, либо писать в него. Каждый процесс представляет собой отдельный поток вычислений (thread) в многопоточной среде. Если процесс пытается читать из канала, в котором нет данных (все данные до этого были им уже прочитаны), то он блокируется. Запись данных в канал не блокирует процесс, так как предполагается, что размер очереди не ограничен.

Кан показал, что такая система обработки данных не зависит от среды

выполнения в том смысле, что порядок обработки входных потоков и генерации выходных потоков данных не зависит от дисциплины планирования процессов. Система будет в любом случае представлять собой непрерывную функцию выходных последовательностей данных от входных. Вычислительный процесс в такой системе будет иметь формальные свойства детерминизма и монотонности (непрерывности, в случае бесконечных входных потоков).

Практическая реализация сетей процессов Кана сопряжена с проблемой ограниченного объема физической памяти в BcC. Модель вычислений подразумевает бесконечный объем доступных ресурсов для хранения данных в каналах и не оговаривает действия среды выполнения при достижении максимально доступного объема. Это приводит к необходимости тщательной верификации конкретной системы для предотвращения переполнения памяти. Кроме того, модель вычислений не гарантирует отсутствие ситуаций блокировки (deadlock) и зависания (livelock), когда два процесса либо не могут прочитать друг у друга данные, либо делают это без предоставления данных другим процессам, которые от них зависят.

Сети процессов Кана являются довольно универсальной моделью вычислений, не дающей формального ответа на многие вопросы реализации, такие как ограниченный объем памяти, невозможность блокировок и зависаний, сложность практической реализации семантики. Тем не менее, эта модель вычислений широко используется при создании компонентов BcC потоковой обработки данных и существуют попытки решения части проблем реализации за счет дальнейшего ее ограничения (например, в Synchronous Dataow возможно статически найти порядок выполнения процессов, при котором каналы данных будут требовать конечный объем памяти).

Модель вычислений хорошо подходит для описания информационного аспекта вычислительного процесса. Она рассматривает входные сигналы как потоки данных. При этом, за рамками модели остаются понятия времени вычислений и моментов их активизации. Это не позволяет адекватно описывать логику управления в рамках сетей с потоками данных.

В сетях процессов Кана процессы взаимодействуют через очереди. Передача данных между процессами осуществляется посредством потоков атомарных объектов (токенов).

Перечислим свойства очередей для данной модели:

- очереди являются однопотоковыми;
- очереди являются однонаправленными;
- чтение токена из очереди всегда приводит к его удалению из очереди;
- глубина очередей равна бесконечности.

Передача и прием каждого токена производится только один раз (т.е. считается, что канал связи между процессами идеальный и токен пропасть не

может). Запись токена в очередь блокирует записывающий процесс, т.к. очередь бесконечной длины. Чтение из очереди блокирует читающий процесс в том случае, когда приемная очередь пуста. Ограничения модели:

- нельзя проверять наличие данных в очереди;
- только один процесс может записывать данные в очередь;
- только один процесс может считывать данные из очереди;
- нельзя разделять память между процессами (организовывать совместные хранилища данных).

Достоинства модели: детерминизм. Недостатки: Невозможность статического планирования.

4.3. Реализация потоковой модели в ОС РВ

Операционная система реального времени – это средство распределения и выделения ресурсов информационно-управляющей системы. Рассмотрим несколько определений термина «ресурсы».

Ресурсы (от франц. Ressource – вспомогательное средство), денежные средства, ценности, запасы, возможности, источники средств, доходов (например, природные ресурсы, экономические ресурсы). Другими словами, ресурсы – это множество «расходных материалов», используемых прикладными процессами для решения целевой задачи. В качестве ресурсов обычно выступают память, процессорное время, устройства ввода-вывода и сеть.

Операционные системы реального времени (ОС РВ) применяются во многих областях. В настоящее время ОС РВ можно найти: в автомобилях и офисной технике, в авиационной и космической технике, в системах вооружения и системах управления технологическими процессами. Рынок информационно-управляющих систем (ИУС) и встроенных (embedded) систем очень динамично развивается. Операционные системы реального времени в настоящее время является неотъемлемой частью большинства ИУС. В самом конце двадцатого столетия возникла тенденция к резкому увеличению капиталовложений в рынок встроенных систем. Это касается как промышленной и военной техники (традиционных экономических ниш подобного рода систем), так и расширяющегося рынка интеллектуальной бытовой техники. Большой спрос на управляющие системы вызвал серьезный прогресс в производстве полупроводников, что в свою очередь, привело к созреванию ряда предпосылок к широкому применению ОС РВ:

- рост быстродействия процессоров для встроенных применений, позволяющий широко использовать весьма ресурсоемкие ОС РВ;
- удешевление элементной базы, в том числе оперативной памяти, FLASH, микропроцессоров, микроконтроллеров и т.д.;

- необходимость интеграции с вычислительными системами общего назначения посредством стандартных сетевых протоколов.

Перечисленные предпосылки в настоящее время делают нерентабельным, а зачастую и просто невозможным разработку целого класса управляющих систем без ОС РВ.

Обобщенная модель ОС РВ

Обобщенная модель ОС РВ (см. рисунок) состоит из системы распределения и управления ресурсами, системы межпроцессных коммуникаций (IPC – interprocess communication), инструментальной системы (инструментального агента) и множества пользовательских процессов. Система межпроцессных коммуникация – это единственное средство для общения процессов друг с другом. В ОС РВ прямое обращение одного процесса к другому, как правило, запрещено. Доступ к ресурсам системы со стороны пользовательских процессов также возможен только централизованно, через единую систему распределения и управления ресурсами. Набор ресурсов определяется способом построения вычислительной системы. Инструментальная система, в силу специфики организации ОС РВ (об особенностях ОС РВ будет говориться далее), играет очень важную роль не только в задаче отладки прикладных процессов, но и в таких задачах как доставка компонентов ОС РВ в целевую систему, тестирование системного ПО, мониторинг процессов, прямое управление ресурсами. Как правило, инструментальные системы ОС РВ значительно сложнее и дороже инструментальных систем ОС общего назначения.

Итак, мы выяснили, что ОС РВ играет роль некого координатора, управляющего процессами, протекающими в ИУС и распределяющего между этими процессами имеющиеся ресурсы. В данном случае возможно сравнение с заведующим складом или библиотекарем (сам задачу не решает, а является распределителем материальных ценностей). Перечислим перечень ресурсов информационно-управляющей системы:

- время процессора (процессоров);
- память различных запоминающих устройств;
- сервисы устройств ввода-вывода;
- сервисы сети.

Классификация ОС РВ

По наличию в ОС РВ инструментальных средств для разработки различают два варианта:

- исполнительская ОС РВ;
- инструментальная ОС РВ.

Исполнительская ОС РВ (Executive, исполнительная) – ОС РВ, не имеющая в своем составе инструментальных средств, достаточных для разработки полноценного программного обеспечения для целевой системы.

Для разработки целевой системы используется кросс-система, в которой разворачиваются все необходимые инструментальные средства. В исполнительской ОС РВ, в лучшем случае, есть только инструментальный агент, отладчик и загрузчик для программного обеспечения.

Примеры исполнительских ОС РВ

- FreeRTOS;
- eCos;
- RTEMS;
- uCOS.

Разработка ПО для таких ОС ведется, как правило, на персональном компьютере, в то время как сама ОС РВ и прикладная задача работают в целевой вычислительной системе, например, в такой как:

- промышленный контроллер ;
- прототипная плата;
- учебный стенд SDK-1.1 или SDK-2.0.

Инструментальная ОС РВ (self hosted, development) – ОС РВ, имеющая в своем составе инструментальные средства, достаточные для разработки полноценного программного обеспечения для целевой системы.

Необходимый комплект инструментальных средств, включающий компилятор, линкер, отладчик и среду разработки присутствует в самой ОС РВ. Инструментальные ОС РВ гораздо больше исполнительских и требуют для работы больших вычислительных ресурсов. В инструментальных ОС РВ существует механизм отчуждения лишних компонент (например, инструментальных) для получения компактной целевой системы.

Например, в ОС Linux с его монолитным ядром для того, чтобы выбросить из ядра лишние компоненты, необходимо произвести перекомпиляцию ядра. В QNX, для получения целевой системы необходимо записать в конфигурационный файл перечень менеджеров, необходимых для работы

системы и поместить исполняемые модули на соответствующий носитель в целевой системе.

Основополагающие компоненты ОС РВ

Общая архитектура ОС РВ будет нами рассматриваться с традиционных позиций, в рамках которых ОС РВ является программой, работающей на классическом процессоре (построенном в рамках модели вычислений Фон-Неймана) и реализующей потоковую модель вычислений.

Системообразующие, самые важные и основополагающие функции ОС РВ сконцентрированы в ядре. К ним обязательно относятся:

- переключатель задач;
- планировщик;
- средства IPC.

Остальные компоненты, в зависимости от типа ядра, могут в него входить или нет.

Ядро ОС РВ

Ядро ОС РВ – центральная часть ОС РВ, содержащая в себе самые важные и сложные компоненты. Четкого определения ядра не существует. В зависимости от того, сколько компонентов входит в состав ядра, можно выделить различные типы систем. Крайними вариантами таких систем являются ОС РВ с микроядерной и монолитной организацией. Ядро ОС РВ, как правило, не контактирует непосредственно с аппаратурой. В качестве дополнительного уровня часто используется HAL.

Существует два диаметрально противоположных подхода к проектированию ядра. Один вариант называется микроядро, другой – монолитное ядро.

В микроядро обычно попадают самые базовые и сложно реализуемые компоненты ОС РВ: переключатель задач, планировщик и средства организации IPC.

По сути, микроядро является надстройкой на машиной Фон-Неймана, позволяющей реализовать в полной мере потоковую модель вычислений. За пределы микроядра попадают системные службы, драйверы и пользовательские процессы.

Из модульной организации следует микроядра следует сравнительная простота компонентов и возможность независимой реализации, отладки и тестирования. К сожалению, простота компонентов еще не говорит о том, что вся система будет простой, так как из теории систем нам известно, что система не есть

простая сумма ее компонентов. Из-за сложного взаимодействия компонентов микроядра друг с другом могут возникнуть ошибки проектирования, приводящие к некорректной работе микроядра и трудноуловимым ошибкам.

При использовании монолитного ядра (monolithic kernel) вся система является единой программой, в которой можно использовать общие структуры данных и организовывать взаимодействие путем простых вызовов процедур [1].

Для добавления новых частей в ядро необходима перекомпиляция, либо реализация механизма оверлеев. Ядро с реализацией оверлеев называют обычно модульным. Оверлей – механизм управления распределением ресурсов, при котором допускается их совместное использование. Как правило, под термином оверлей понимается механизм повторного использования блоков памяти данных и кода в вычислительных системах, работающих в рамках модели вычислений Фон-Неймана.

К достоинствам монолитного ядра по сравнению с микроядром можно отнести высокую скорость работы и упрощение взаимодействия между частями ядра. К недостаткам монолитного ядра относятся сложность отладки и тестирования, более низкая, по сравнению с микроядром надежность, так как у всех частей ОС РВ общие структуры данных и общее адресное пространство.

В своей книге «Just for fun» Линус Торвальс так отзывается о особенностях микроядра. «Теоретически необходимость микроядра обосновывается следующим образом. Операционные системы сложны. Для их упрощения применяется модульный подход. Вся соль микроядра в том, чтобы оставить у ядра, которое является основой основ, как можно меньше функций. Его главная задача – обмен информацией. А все возможности компьютера реализуются в виде сервисов, которые обеспечивают коммуникационные каналы микроядра. Предполагается, что вы разбиваете проблемы на такие мелкие части, что вся сложность пропадает.

Мне это казалось глупым. Да, каждая отдельная часть получается простой. Но при этом их взаимодействие становится гораздо более сложным, чем при включении ряда сервисов в состав ядра, как это сделано в Linux. Представьте себе человеческий мозг. Каждая его составляющая проста, но их взаимодействие превращает мозг в очень сложную систему. В этом-то все и дело: целое больше частей. Если взять проблему, разделить ее пополам и сказать, что каждая половинка вполовину проще, то при этом вы игнорируете сложность интерфейса между половинками. Сторонники микроядра предлагали разбить ядро на пятьдесят независимых частей так, чтобы каждая часть была в пятьдесят раз проще. Они умалчивали о том, что взаимодействие между частями окажется сложнее исходной системы, при том, что и части сами по себе не будут элементарными.

Это самое главное возражение против микроядра. Простота, обеспечиваемая микроядром, является мнимой.»

На самом деле, решать, как реализовывать ядро нужно по ситуации. В каких-то ситуациях может быть выгодным монолитное ядро, в каких-то – микроядро. В любом случае, при проектировании системы нужно взвешивать все плюсы и минусы разных вариантов.

Обзор компонентов ядра ОС РВ

Переключатель задач ОС РВ

Переключатель задач ОС РВ – один из самых распространенных механизмов организации потоковой модели вычислений в системах на базе машины Фон-Неймана. Как известно, сама по себе модель Фон-Неймана не имеет удобных средств для создания многозадачности. Для расширения исходной модели впоследствии были добавлены такие механизмы как прерывания, команды разрешения и запрещения прерываний, стек возвратов и таймер. Набор дополнительных средств позволяет реализовывать несколько удобных способов организации потоковой модели вычислений.

Классифицируя способы переключения задач можно выделить два основных критерия: причину переключения и способность прерывания одной задачи другой.

Причин переключения всего две: время (time triggered) или наличие события (event triggered). Первый вариант переключения осуществляется по специально созданному расписанию. Второй позволяет вызывать задачи как ответ на возникающее событие.

По способности прерывать одну задачу другой можно выделить согласующие переключатели, в которых не происходит прерывания задач и они выполняются всегда до конца и вытесняющие переключатели, которые умеют прерывать задачу.

В самом простом варианте параллельные процессы создаются путём написания обработчиков прерываний. Очень грубо и упрощенно, процесс работы выглядит примерно следующим образом.

- В контроллер прерываний приходит запрос на прерывание.
- Контроллер прерываний прекращает выполнение текущей последовательности команд. На стеке сохраняется адрес следующей, еще не исполненной команды.
- Одним из способов (например, из таблицы векторов прерываний) извлекается адрес обработчика прерывания и ему передается управление.
- Внутри обработчика прерывания происходит выполнение последовательности команд, решающих задачу, связанную с целью данного прерывания.
- По окончании работы обработчика, последней командой исполняется

команда возврата из прерывания, из стека изымается адрес возврата и осуществляется переход на этот адрес.
- Начинается выполнение прерванной ранее программы.

Как правило, контроллеры прерываний имеют приоритетную систему выбора прерывания, обработчик которого надо вызвать в первую очередь. Суть приоритета в том, что это числовая оценка важности задачи. Чем выше приоритет — тем важнее задача, и тем раньше ее надо выполнить. Казалось бы — вот хороший и простой механизм, позволяющий получить псевдопараллельное исполнение программ на базе Фон-Нейманновской архитектуры. Однако не все так просто, как кажется. Когда задач много, становится достаточно трудно оценить, будет выполнена та или иная задача вовремя или нет. Какие есть проблемы у данного подхода? Если внутри обработчика прерывания запретить прерывания, то все остальные обработчики не смогут выполняться, до тех пор, пока не закончится более высоко приоритетное прерывание. Если выключить запрет прерываний внутри обработчиков могут возникнуть проблемы со стеком и повторными вхождениями, а главное, будет очень трудно предсказать поведение программы при достаточно большом количестве прерываний. Для структурирования процесса, придумали переключатель задач. Суть переключателя состоит в выделении каждой задачи кванта времени определенной длительности. Интервалы между переключениями задач задаются с помощью таймера. Внутри же обработчика прерываний от таймера находится переключатель задач. Как это работает? Примерно следующим образом.

- Таймер переполняется и выставляет запрос на прерывание.
- Контроллер прерываний прерывает работу текущей программы.
- Адрес следующей невыполненной команды и содержимое регистров сохраняется в стеке.
- Переключатель задач запоминает указатель стека в контексте i-го процесса.
- Далее, переключатель задач выбирает из массива значение указателя стека для процесса i+1 и устанавливает его.
- Выполняется команда возврата из прерывания. Мы попали в другой процесс.

В описанной выше схеме процессы получают одинаковые кванты времени и их выполнение имеет одинаковый приоритет. Для управления приоритетом задач в таких системах используют специальные планировщики.

В качестве простого примера рекомендую посмотреть исходные тексты переключателя задач операционной системы FreeRTOS. Эта простая и компактная операционная система с открытыми исходными текстами распространяется свободно и может быть использована даже в коммерческих проектах. Исходные тексты доступны на сайте http://www.freertos.org/.

Приведенный пример переключателя задач принадлежит FreeRTOS версии 5.0.3 и взят из следующего каталога исходных текстов:

FreeRTOS\Source\portable\SDCC\Cygnal

```
// ----------------------------------------------------------
// Переключатель задач FreeRTOS
// (порт для ядра MCS51, компилятор SDCC)
// ----------------------------------------------------------

void vTimer2ISR( void ) interrupt 5 _naked
{
    portSAVE_CONTEXT();
    // Сохранение контекста в стеке
    // (push regs)

     portCOPY_STACK_TO_XRAM();
    // Копирование стека во внешнее ОЗУ

     vTaskIncrementTick();
    // Инкремент системных часов

     vTaskSwitchContext();
    // Выбор очередной задачи из списка

     portCLEAR_INTERRUPT_FLAG();
    // Корректное завершение прерывания,
    // сброс флагов

     portCOPY_XRAM_TO_STACK();
    // Перенос контекста из внешнего ОЗУ
    // в стек

     portRESTORE_CONTEXT();
    // Восстановление контекста         // (pop regs)
}
```

Функция переключателя задач vTimer2ISR реализована как обработчик прерывания от таймера. Исходный текст функции в полном виде можно посмотреть в файле port.c. В файле task.h можно посмотреть описания системных вызовов.

Реализация планирования задач в ОС РВ

Планировщик (диспетчер) – система для планомерного выделения ресурсов в рамках определенной политики (методики). Суть планировщика – формирование управляющих сигналов в ответ на изменения, возникающие в процессе вычислений. Планировщики, реализующие механизм взаимного исключения, называют мониторами или арбитрами.

В рамках ОС РВ, планировщик, как правило, выделяет только один ресурс – ресурс процессорного времени.

В ОС РВ планировщик используют для выделении из всех задач самой важной и выделения ей необходимых для работы ресурсов. Обычно под планировщиком понимается программа, определяющая порядок вызова задач и длительность кванта времени в операционных системах с псевдомногозадачностью. Таким образом, в обычных однопроцессорных операционных системах планировщик планирует такой ресурс как процессорное время. Планировщик работает в соответствии с заданной методикой, так называемым алгоритмом планирования.

В ОС РВ планировщик не является монопольным распределителем ресурсов. Кроме планировщика, к вопросам управления ресурсом процессорного времени (процессами) относятся:

- средства IPC;
- средства синхронизации и взаимного исключения (семафоры и мьютексы, критические секции).

Такое разделение полномочий и отдача сложных средств управления ресурсами на откуп прикладным программистам обычно приводит к отсутствию како-либо формализации вычислительного процесса и к большому количеству ошибок.

Межпроцессное взаимодействие

Межпроцессное взаимодействие, IPC (Inter-process communication) – набор механизмов для взаимодействия между процессами, работающих в рамках потоковой модели вычислений.

В IPC входят механизмы обмена данными и управления процессами. Под управлением понимаются такие действия, как запуск и остановка процессов с целью синхронизации.

Механизмы обмена данными предназначены только для передачи данных от одного процесса к другому. Управления состоянием процесса в таких каналах нет. Для примера можно рассмотреть сети процессов Кана.

1. Каналы без блокировки.
2. Очереди (FIFO), потоковый обмен данными, сокеты.
3. Пакетная пересылка, почтовые ящики (mail box).
4. Общая память.

Механизмы управления не передают из процесса в процесс каких-либо данных. Семафоры, мониторы и критические секции позволяют осуществлять управление процессами.

- Семафоры.

- Мониторы.
- Критические секции.

Гибридные механизмы позволяют не только передавать данные между процессами, но и запускать и останавливать процессы.

- Каналы с блокировкой.
- Сигналы.

4.4. Взаимное исключение

Почему нельзя использовать простые флаги при организации совместного доступа нескольких процессов или потоков к одному ресурсу? Зачем нужны алгоритмы Петерсона, Деккера и семафорфы, если все обычно используют простые флаги?

1. Операторы языков высокого уровня обычно транслируются не в одну а в несколько команд.
2. Переключатель процессов может прервать процесс на любой команде.
3. Процесс 1 проверил флаг критической секции. Флаг свободен.
4. Процесс 1 начал модифицировать флаг. Процесс 1 может считать, что он модифицировал флаг критической секции. На самом деле его прервал процесс 2.
5. Процесс 2 изменяет флаг критической секции и начинает работать с данными.
6. Процесс 1 прерывает процесс 2. Он уже проверил флаг и считает, что он свободен. Процесс 1 модифицирует флаг и начинает работать с критическими данными. Возникает конфликт.
7. Для получения атомарности, используют два основных метода: реализация неделимых операций TEST_AND_SET и CLEAR_TEST_AND_SET (реализация двоичного семафора Дейкстры на уровне системы команд ЦП).
8. Использование аппаратных прерываний.

Взаимное исключение (Mutual exclusion) – механизм, гарантирующий, что только один процесс производит некую специфическую деятельность. Все остальные процессы исключены из выполнения этой деятельности. Взаимное исключение относится к синхронизации конкуренции.

Термин «взаимное исключение» обычно используют в контексте программного обеспечения в рамках операционных систем, в частности, в рамках ОС РВ (при использовании потоковой модели вычислений). На самом деле, механизм взаимного исключения универсален и его можно применять не только в программном, но и в аппаратном обеспечении. Приведем примеры:

- арбитр системной шины;
- мьютекс для организации IPC между процессорами NIOS в среде Quartus.

Мьютекс – средство взаимного исключения, двоичный семафор. Переменная, хранящаяся внутри мьютекса, может принимать значения 0 и 1.

Методы реализации взаимного исключения, классификация

По местонахождению механизма управления процессами, можно выделить два способа организации взаимного исключения:

- внешние;
- внутренние (пользовательские).

Внешние методы предполагают управление прикладными процессами снаружи. Механизм управления процессами может быть встроен в среду, создающую потоковую модель вычислений, а может быть реализован в виде отдельного, системного процесса, имеющего право управлять прикладными процессами.

Первый вариант управления обычно реализуют в виде планировщика, второй – в виде монитора.

Монитор или арбитр – средство для организации арбитража, один из механизмов реализации взаимного исключения, позволяющий нескольким процессам обращаться к одному ресурсу. В отличие от семафора, монитор сам решает, когда и кому предоставить ресурс для использования. Такой подход позволяет избавиться от проблем со взаимной блокировкой.

Термин монитор обычно используют в контексте программной реализации взаимного исключения в рамках операционных систем. Термин арбитр чаще применяют в контексте цифровых устройств. Как правило, арбитры используются в различных системных и внутрисистемных интерфейсах.

Если существует формальный алгоритм планирования, удовлетворяющий поставленной задаче и условиям работы системы, то работа в реальном времени будет возможна.

К недостатку метода можно отнести сложность реализации и необходимость в изначальном планировании способов решения задачи. Так как не все задачи формализованы полностью, например, задача планирования апериодических задач, не для всех случаев годится внешний метод взаимного исключения.

Обычно у программистов внешние методы не пользуются особой популярностью.

Суть пользовательского метода в том, что низкоуровневое управление процессами производится не централизованно (например, из ядра ОС РВ, а из

пользовательского процесса). Преимущество метода в простоте реализации.

Использование таких методов чаще всего приводит к путанице, усложнению программы и уменьшению её надежности. Из-за большой сложности поведения программы (часто неопределенности поведения) доказать, что она будет работать корректно в режиме реального времени фактически невозможно. Применение метода носит эвристический характер.

В методах организации взаимного исключения действую следующие приемы.

- Функции входа и выхода их критической секции делаются поэтапными (в критической секции используется больше одной переменной, используется активное ожидание):
    - лгоритм Деккера;
    - лгоритм Петерсона;
    - лгоритм пекарня;
- Функции входа и выхода в критическую секцию делаются атомарными (неделимыми), используются механизмы управления прерываниями и переключением задач. Этот метод используется значительно чаще предыдущего:
    - организация критической секции за счет запрета и разрешения прерываний (переключения процессов);
    - семафоры Дейкстры.

Алгоритм Петерсона – программная реализация механизма взаимного исключения без запрещения прерываний. Алгоритм Петерсона имеет смысл в системах на базе модели вычислений Фон-Неймана. Алгоритм Петерсона предложен в 1981 году Гарри Петерсоном из университета Рочестер (США). В основу алгоритма Петерсона лег алгоритм Деккера.

Алгоритм имеет следующие ограничения.

- Алгоритм рассчитан только на 2 процесса (от этого ограничения свободен алгоритм пекарня (Bakery algorithm)).
- При ожидании ресурса процессы не снимаются с очереди на обслуживание и впустую тратят процессорное время (активное ожидание).

Алгоритм Петерсона учитывает отсутствие атомарности в операциях чтения и записи переменных и может применяться без использования команд

управления прерываниями.

Алгоритм работает на использовании двух переменных: у каждого процесса есть собственная переменная **flag[i]** и общая переменная **turn**. Все переменные хранятся в общей для обоих процессов памяти. В переменной **flag** запоминается факт захвата ресурса, в переменной **turn** – номер захватившего ресурс процесса.

При исполнении пролога критической секции процесс *Pi* заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

```
flag[0]   = 0
flag[1]   = 0
turn      = 0

P0: flag[0] = 1                  P1: flag[1] = 1
turn = 1                         turn = 0
while( flag[1] && turn == 1 );   while( flag[0] && turn == 0);
// ждем                          // ждем
// начало критической секции     // начало критической секции
...                              ...
// конец критической секции      // конец критической секции
flag[0] = 0                          flag[1] = 0
```

В начале процесс устанавливает флаг занятости, затем – номер процесса соседа. После этого каждый из процессов входит в цикл ожидания. Выход из цикла происходит, если флаг занятости установлен и номер процесса соответствует номеру соседа.

Еще один вариант реализации алгоритма Петерсона:

```
void mut_excl(int me /* 0 or 1 */)
{
    static int loser;
    static int interested[2] = {0, 0};
    int other; /* local variable */

    other = 1 - me;
    interested[me] = 1;
    loser = me;
    while (loser == me && interested[other])
        ;

    /* critical section */
    interested[me] = 0;
}
```

Алгоритм Деккера – программная реализация механизма взаимного исключения без запрещения прерываний. Алгоритм имеет смысл в системах на базе модели вычислений Фон-Неймана. Алгоритм Деккера усовершенствован Петерсоном (см. алгоритм Петерсона).

Алгоритм имеет следующие ограничения.

- Алгоритм рассчитан только на 2 процесса (от этого ограничения свободен алгоритм пекарня (Bakery algorithm)).
- При ожидании ресурса, процессы не снимаются с очереди на обслуживание и впустую тратят процессорное время (активное ожидание).

Принцип работы аналогичен используемому в алгоритме Петерсона.

- Если два процесса попытаются попасть в критическую секцию одновременно, алгоритм позволит войти только одному процессу, номер которого соответствует числу в переменной **turn**.
- Если один из процессов уже находится в критической секции, другой процесс будет находится в ожидании.

Необходимо заметить, что использование подобных алгоритмов в ОС РВ нежелательно, так как заблокированный процесс не снимается с обслуживания и напрасно расходует процессорное время.

```
f0 := false
f1 := false
turn := 0   // or 1

p0:
  f0 := true
  while f1 {
      if turn ≠ 0 {
          f0 := false
          while turn ≠ 0 {
          }
          f0 := true
       }
  }

// ждем
// начало критической секции
...
// конец критической секции
    turn := 1
    f0 := false
```

```
p1:
  f1 := true
   while f0 {
       if turn ≠ 1 {
           f1 := false
           while turn ≠ 1 {
           }
           f1 := true
       }
    }

// ждем
// начало критической секции
     ...
// конец критической секции
         turn := 0
         f1 := false
```

Алгоритм пекарня – программная реализация механизма взаимного исключения без запрещения прерываний. Алгоритм пекарня имеет смысл в системах на базе модели вычислений Фон-Неймана. В отличии от алгоритма Деккера и Петерсона отсутствует ограничение на число процессов.

Изобретен Лесли Лампортом (Leslie Lamport). Приоритет процессов

задается случайным образом, алгоритм плохо предсказуем и практически не применим для решения задач реального времени. Кроме того, при ожидании ресурса процессы не снимаются с очереди на обслуживание и впустую тратят процессорное время (активное ожидание).

Лампорт предложил свой алгоритм по аналогии с пекарней (булочной). У нас аналогичная система существует в системах быстрого питания или в поликлиниках. При входе каждый клиент получает табличку с уникальным номером. В один момент времени производится обслуживание только одного клиента. При выходе клиент табличку отдает. Первым обслуживается вошедший клиент, имеющий минимальный номер. Так как операции не атомарные, одинаковый номер могут получить несколько клиентов. В таком случае можно выбрать приоритетность клиентов по имени процесса.

```
// Определение и начальная инициализация глобальных переменных
// Переменные доступны всем процессам
Entering: array [1..N] of bool = {false};
Number: array [1..N] of integer = {0};
// ==============================
// Вход в критическую секцию
// i - номер процесса
// ==============================
lock(integer i)
{
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[N]);
    Entering[i] = false;
    for (j = 1; j <= N; j++)
    {
        // Ждем
        while (Entering[j]);
        // Ждем
        // while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i)));
    }
}
// ==============================
// Выход из критической секции
// i - номер процесса
// ==============================
unlock(integer i)
{
    Number[i] = 0;
}
// ==============================
// Пример использования
// i - номер процесса
// ==============================
Process_example(integer i)
{
    while (true)
    {
        lock(i);
        // Критическая секция
        unlock(i);
        // Выход из критической секции
    }
```

```
}
```

Критическая секция

Критическая секция (Critical section) – это часть программы, в работу которой не может вмешаться другой процесс. Понятие "критическая секция" имеет смысл в тех системах, где возможно прерывание какого либо действия. Как правило, критические секции используются в системах, сделанных на базе модели вычислений Фон-Неймана и Process Network. Другими словами, критические секции используются при программировании классических микроконтроллеров и микропроцессоров, при использовании ОС РВ или прерываний.

Критическая секция – один из механизмов управления процессами.

В ОС РВ к процессам, использующим взаимное исключение, предъявляются следующие требования:

- Процесс попавший в критическую секцию не должен быть заблокирован навсегда (голод).
- Если несколько процессов вводят критические секции, то процессы должны рано или поздно из них выйти (взаимная блокировка).

Семафор

Семафор – один из механизмов реализации взаимного исключения, позволяющий нескольким процессам обращаться к одному ресурсу. Идея семафоров предложена нидерландским ученым Дейкстрой в 1965 г. Ввиду того, что семафоры управляются извне, самими процессами, которым нужен выделяемый ресурс, при использовании семафоров возможен ряд проблем:

- взаимная блокировка;
- утечка семафоров;
- проблема синхронизации процедур семафоров.

Взаимная блокировка, тупик, клинч, дедлок (deadlock) – ситуация, которая может возникнуть в системе, выполненной на безе модели вычислений "сеть процессов", при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, захваченных этими процессами.

Рассмотрим пример взаимной блокировки. Пусть имеются 2 процесса A и B, которым перед началом работы предоставлены ресурсы P1 и P2, соответственно. В какой-то момент времени процессу A понадобился P2, а процессу B – P1, но они их не получат, т.к. удерживаются предыдущими

процессами.

Предотвращение взаимной блокировки.

1. Прежде чем процесс начнет свою работу, ему должны быть предоставлены все требуемые ресурсы.
2. В том случае, если во время работы ему понадобился дополнительный ресурс, ему необходимо возвратить все ранее выделенные ресурсы ОС и затем запросить все требуемые ресурсы с этим дополнительным ресурсом.

Утечка семафоров – проблема, возникающая при использовании семафоров, когда операция захвата ресурса производится, а операция освобождения – нет. В результате это проблемы ресурс оказывается захвачен навсегда.

Проблема синхронизации процедур самого семафора выглядит следующим образом. Возможна следующая ситуация: два процесса ждут освобождения семафора. После того, как семафор освободился, первый процесс «узнаёт» об этом, но не успевает увеличить счётчик, так как управление передаётся второму процессу. Второй процесс тоже узнаёт об освобождении семафора, увеличивает счётчик и входит в защищённый участок кода. Тут управление передаётся первому процессу, последний ещё раз увеличивает счётчик и тоже входит в защищённый участок кода. В итоге имеем превышение разрешённого числа процессов.

Данная проблема не имеет алгоритмического решения. Она разрешается либо размещением процедуры ожидания в критической секции, в которой не разрешается переключение с процесса на процесс, либо программистскими приёмами, вроде осуществления проверки флага и его увеличения с помощью одной машинной команды. [wikipedia]

В программном обеспечении семафор обычно реализуется в виде блокирующей переменной s, изменение состояния которой производится одним (атомарным) действием (т.е. изменение нельзя прервать) с использованием функций **P(s)** и **V(s**).

В процессе инициализации в семафор записывается начальное значение.

```
s = init_value;
```

Существует два действия над семафорами: P (s) – занять семафор V(s) – освободить семафор. При попадании на занятый семафор процесс блокируется до тех пор, пока другой процесс не освободит семафор.

Функция P(s):

```
if( s == 0)
        Заблокировать_текущий_процесс();
else
        s--;
```

Функция V(s):

```
if( s == 0)
        разблокировать_один_из_процессов();
else
        s++;
```

В сетях Ethernet, имеющих способ доступа CSMA/CD, семафором является *сама шина*. Этот семафор может хранить только один бит информации, так как шина либо занята, либо нет. Таким образом, можно сказать, что в Ethernet для разделения ресурса шины используется бинарный семафор или мьютекс.

В цифровой технике в явном виде семафор не используется из-за недостатков, присущих семафорам. Вместо семафора в аппаратном обеспечении обычно используют арбитр.

4.5. Обзор ОС РВ

ОС РВ FreeRTOS

FreeRTOS – портативная операционная система реального времени (ОСРВ) с открытым исходным кодом для встроенных систем. Спектр поддерживаемых ею аппаратных и средств разработки достаточно широк.

Основные особенности ОС:

- построена таким образом, чтобы быть небольшой и простой в использовании;
- переносимая структура кода, преимущественно написанная на С;
- отсутствие программных ограничений на число задач, которые могут быть созданы;
- отсутствие программных ограничений на количество приоритетов, которые могут быть использованы;
- не налагается ограничений на назначение приоритетов – более чем одной задаче может быть присвоен одинаковый приоритет;
- очереди и семафоры для связи и синхронизации между задачами или задачами и прерываниями;
- мьютексы с наследованием приоритета;
- позволяет выбрать политику переключения задач:

1. вытесняющую;
2. кооперативную.

Организация исходного кода

Платформо-независимый код находится в 4 файлах в каталоге source. Каждый файл содержит реализацию одной из четырех подсистем ОС. Аппаратно зависимый код (порты на микроконтроллеры и компьютеры) состоит из 2-х файлов. В этих файлах происходит определение переносимых целочисленных/вещественных типов, типа элементов стека, базового целочисленного типа. К платформо-зависимым также относятся некоторые функции системы, такие как:

- разрешение/запрещение прерываний;
- сохранение/восстановление контекста задачи;
- инициализация таймера;
- инициализация стека;
- обработчик прерываний от таймера.

The RTOS tick interrupt is generated from timer 2.

Каждая задача должна иметь свой собственный стек. Однако архитектура 8051 позволяет располагать стек только в небольшой внутренней памяти RAM. Поэтому при переключении задач стек необходимо копировать из маленькой RAM в XRAM и наоборот. Этот процесс полностью ложится на программиста.

Конфигурация и детали использования

Файл FreeRTOSConfig.h содержит все настройки ОС. Перечислим основные из них. Константа configSTACK_START содержит адрес начала стека. Она определена корректно для демо-приложения, но требует обновления для вашего приложения, если в нем присутствуют переменные объявленные с модификатором data. Значение configSTACK_START может быть получено из .mem файла, создаваемого компилятором SDCC во время компиляции приложения. Например, если в .mem файл содержит строку "Stack starts at: 0x0e (sp set to 0x0d) with 242 bytes available", то переменной configSTACK_START должно быть присвоено значение 0x0E. Для переключения режимов многозадачности (вытесняющая и кооперативная) используется переменная configUSE_PREEMPTION. Если она установлена в 1, то используется вытесняющая многозадачность, если в 0, то кооперативная.

Менеджер памяти

Одной из основных задач ОС является управление памятью. Ядро выделяет память для своих нужд только в следующих случаях: создание задачи, создание очереди, создание семафора. Стандартные функции языка C malloc() и free() конечно могут быть использованы для этой цели, но:

- они не всегда поддерживаются встроенными системами;
- они занимают достаточно ограниченное пространство кода;

- они ненадежны;
- они не детерминированы (время выполнения этих функций отличается от вызова к вызову).

Вследствие всего этого реализация менеджера памяти дается на откуп ОС. Функции выделения и освобождения памяти могут быть специфичны для каждой системы, поэтому относятся к аппаратно-зависимой части ОС.

Интерфейс менеджера памяти pvPortMalloc() выделяет блок памяти требуемого размера. vPortFree() освобождает выделенный блок.

В комплекте с ОС поставляется три примера реализации управления памятью, расположенных в source\portable\MemMang. Все задачи используют общий пул памяти. Вызовы потокобезопасны. Размер пулов задается configTOTAL_HEAP_SIZE.

Управление задачами

ОС позволяет организовать несколько потоков выполнения задач. В контекст задачи входит содержимое регистров и стека. Вся остальная память разделяется задачами. Каждая задача в любой момент времени может находиться в одном из следующих состояний:

- Running – исполняется на процессоре;
- Ready – готова занять процессор при возможности;
- Blocked – ожидает наступления события (список: delay, queue, semaphore);
- Suspended – «поставлена на паузу».

Только одна задача единовременно может быть в состоянии Running. Все задачи со статусом Ready имеют не больший приоритет. У каждой задачи есть приоритет, выраженный целым неотрицательным числом. Чем больше число – тем выше приоритет. Процессор всегда получает задача с наибольшим приоритетом. На рисунке (см. рис. 1) изображен граф переходов между состояниями задачи.

Создание и удаление задач

Тело задачи должно иметь следующую структуру.

```
void vATaskFunction (void *pvParameters)
{
    for (;;)
    {
        // -- Task application code here. --
    }
}
```

Задачи создаются вызовом xTaskCreate() и удаляются вызовом

vTaskDelete().

```
portBASE_TYPE xTaskCreate(
                            pdTASK_CODE pvTaskCode,
                            const portCHAR * const pcName,
                            unsigned portSHORT usStackDepth,
                            void *pvParameters,
                            unsigned portBASE_TYPE uxPriority,
                            xTaskHandle *pvCreatedTask
                           );
```

Создает новую задачу и добавляет ее в список задач готовых к запуску.

- PvTaskCode – указатель на задачу. Задача должна быть объявлена так, чтобы из нее не происходило выхода.
- PcName – имя задачи, главным образом используется для отладки

```
void vTaskDelete( xTaskHandle pxTask );
```

Задача удаляется из всех список состояний и событий ОС.

- PxTask – дескриптор задачи для удаления. Сбрасывается в NULL если после вызова задача была успешно удалена.

Старт системы

После создания задач пользователь должен запустить планировщик. Это осуществляется с помощью системного вызова:

```
void vTaskStartScheduler( void );
```

Возврата из этой функции не происходит.

ОС РВ eCos

eCOS 2 – это операционная система реального времени ОС РВ с открытыми исходными кодами, распространяющаяся на условиях лицензии GPL. Она имееет следующие характеристики.

- Мощную конфигурационную систему с графическим или консольным интерфейсом.
- Полнофункциональное, расширяемое и конфигурируемое ядро реального времени, которое обеспечивает планирование, синхронизацию и взаимодействие потоков, а также осуществляет управление ресурсами аппаратуры (прерываниями, памятью и т.д.).
- Ядро поддерживает вытесняющую или невытесняющую многозадачность

в зависимости от настроек.

- Слой абстракции оборудования (HAL), который скрывает специфические особенности конкретного процессора и платформы, в результате чего ядро и многие другие компоненты могут быть реализованы в переносимом виде
- Поддержку программных интерфейсов (API) – μITRON и POSIX. Кроме этого **eCOS** также содержит полнофункциональную, потокобезопасную библиотеку ISO C и математическую библиотеку.
- Поддержку широко диапазона устройств, включающего последовательные устройства, Ethernet контроллеры, Flash память, PCMCIA и USB устройства.
- Полностью функциональный стек TCP/IP поддерживающий IP, IPv6, ICMP, UDP, TCP, SNMP, HTTP, TFTP и FTP поверх Ethernet соединения.
- RedBoot – монитор ПЗУ, который использует eCOS HAL, позволяющий производить загрузку и отладку приложений на целевой платформе через последовательный канал или Ethernet соединение.

eCOS включает в себя множество тестовых программ, которые проверяют корректность поведения различных элементов ОСРВ и служат примерами ее использования. Обширная документация по eCOS доступна на сайте [1].

ОС РВ **eCOS** имеет модульную структуру, которая состоит из следующих модулей.

1. Ядра eCOS.
2. Слоя абстракции оборудования (HAL).
3. Математической и C библиотеки.
4. Пакет В/В (драйверы устройств).
5. Инфраструктуры поддержки собственной файловой системы.
6. Библиотеки поддержки Flash, CAN, PCI, USB и т.д.
7. Слоев совместимости со стандартами POSIX и μITRON.
8. Поддержка собственного стека TCP/IP, а также стека операционных систем FreeBSD и OpenBSD.
9. Поддержка DNS и IPSEC.
10. Наличие реализации собственного небольшого HTTP сервера – ATHTTP, FTP и SNTP клиента

Задача в ОС РВ eCOS имеет формат:

```
void thread_entry_function(cyg_addrword_t data) ,
```

При этом, в отличие от uC/OS-II, задача может возвратиться из своей функции, но при возврате будет вызван метод `cyg_thread_exit`.

Задача может быть написана на C и C++.

Для осуществления необходимой для приложения начальной инициализации можно определить следующую функцию, управление которой будет передано до старта системы:

```
void cyg_user_start(void) ,
```

но, если приложение на C++, то уже был произведен вызов конструкторов для статических объектов.

Большая часть eCOS написана на C, хотя есть части системы на C++. Основу конфигурационной системы составляют скрипты и условия, написанные на языке определения компонентов (CDL).

В 2002 году из eCOS 2 выделился платный вариант этой операционной системы реального времени и сопутствующего инструментария, получивший название eCosPro, который в настоящее время за определенную сумму предоставляется фирмой eCosCentric.

ОС РВ QNX

QNX – коммерческая POSIX-совместимая, инструментальная, микроядерная операционная система реального времени (ОС РВ), предназначенная преимущественно для встроенных систем.

Как микроядерная операционная система, QNX основана на идее работы основной части своих компонентов как небольших задач, называемых серверами. Это отличает её от традиционных монолитных ядер, в которых ядро операционной системы – одна большая программа, состоящая из большого количества частей, каждая со своими особенностями. Использование микроядра в QNX позволяет пользователям (разработчикам) отключить любую ненужную им функциональность, не изменяя ядро. Вместо этого, можно просто не запускать определённый процесс.

Система достаточно небольшая, чтобы в минимальной комплектации уместиться на одну дискету, вместе с этим она считается очень быстрой и должным образом законченной (практически не содержащей ошибок).

QNX Neutrino, выпущенная в 2001 году, перенесена на многие платформы и сейчас способна работать практически на любом современном процессоре, используемом на рынке встраиваемых систем. Среди этих платформ присутствуют семейства x86, MIPS, PowerPC, а также специализированные семейства процессоров, такие как SH-4, ARM, StrongARM и xScale. Версия для некоммерческого использования доступна для скачивания на вебсайте разработчика.

В 1980 году студенты канадского Университета Ватерлоо Гордон Белл и Дэн Додж закончили изучение базового курса по разработке операционных систем, в ходе которого они создали основу ядра, способного работать в реальном времени. Разработчики были убеждены, что в их продукте была коммерческая потребность, и переехали в Канату в штате Онтарио (город высоких технологий, иногда это место называют северной Силиконовой долиной Канады) и основали компанию Quantum Software Systems. В 1982 году была выпущена первая версия QNX, работающая на платформе Intel 8088.

Одно из первых применений QNX, получивших широкое распространение, не относилось ко встраиваемым системам, оно было выбрано для собственного компьютерного проекта Министерства Образования Онтарио, Unisys ICON. В те годы QNX использовалось в основном только для больших проектов, так как ядро, имеющее размер 44 килобайта, было слишком большим, чтобы работать на однокристальных чипах того времени. В середине 80-х годов была выпущена QNX2. Система получила завидную репутацию за надёжность, и приобрела широкое распространение для управления промышленными машинами. QNX2 и сейчас иногда применяется во многих ответственных системах.

В середине 1990-х в Quantum поняли, что на рынке быстро завоёвывает популярность POSIX, и решили переписать ядро, чтобы оно было более совместимым на низком уровне. Так появилась QNX4. Она была доступна со встраиваемой графической подсистемой, названной Photon microGUI и портированной под QNX версией X Window system. Перенесение программ в QNX4 из операционных систем, основанных на Unix, стало намного проще, также были убраны многие из «причуд» более ранних версий. В начале 90-х компания была переименована в QNX Software Systems, чтобы избежать путаницы с другими компаниями, в первую очередь с производителем жёстких дисков, имеющим такое же имя.

В конце 1990-х было решено создать операционную систему, максимально совместимую с Linux, в то же время сохранив микроядерную архитектуру. Результатом этих разработок стала QNX Neutrino, выпущенная в 2001 году. Эта версия поставляется вместе с Momentics, средой разработки программного обеспечения (IDE), основанной на Eclipse IDE, различными утилитами GNU и програмным обеспечением, ориентированным на Интернет: веб-браузерами Voyager и Mozilla и веб-сервером. В отличие от предшествующих версий, работавших только в PC-совместимых архитектурах, QNX6 легко адаптируется к процессорным платам практически любой конфигурации. Кроме того, особое внимание было уделено проработке архитектуры, с тем, чтобы её можно было эффективно масштабировать: как вверх (добавляя новые сервисы и расширяя функциональность), так и вниз (урезая функциональность, чтобы «втиснуться» в ограниченные ресурсы).

Иными словами, QNX6 можно установить там, где QNX4 не уместилась бы. Также в QNX6 все драйверы были приведены к единой модели, и все

интерфейсы стали открытыми.

В 2004 году QSS была куплена компанией Harman, весьма далекой от информационных технологий и ОСРВ, в частности. В результате QSS превратилась из независимого разработчика и поставщика решений для широкого спектра применений в отдел ИТ компании Harman.

После 2004 года интерес в мире к QNX начал угасать. QNX сохранила популярность в основном в России и СНГ благодаря неимоверным усилиям дистрибьюторов, а так же в Германии, по причине агрессивной рекламы. 12 сентября 2007 года компания QNX Software Systems обьявила о том, что исходный код ОС QNX Neutrino будет открыт, но для коммерческого использования QNX Neutrino необходимо приобрести лицензию.

Архитектура QNX

- Микроядро QNX, координирующее менеджеров системы.
- Микроядро.
- Менеджер процессов.
- Пространство ввода-вывода.
- Менеджер файловой системы.
- Менеджер устройств.
- Менеджер сети.
- Photon microGUI.

QNX состоит из маленького ядра, отвечающего за группу взаимодействующих процессов. Как показано на следующих иллюстрациях, его структура напоминает скорее «команду», чем иерархию, поскольку несколько игроков равного ранга взаимодействуют друг с другом и с «защитником» – ядром.

Микроядро

Ядро – основа любой операционной системы. В некоторых системах ядро включает так много функций, что является полной операционной системой. Но микроядро QNX – действительно является ядром. Во-первых, как ядро операционной системы реального времени, микроядро QNX очень маленькое. Во-вторых, оно обеспечивает только две необходимых функции:

- передачу сообщений (message passing) – микроядро обрабатывает поток всех сообщений всех процессов во всей системе;

- планирование (scheduling) – планировщик является частью микроядра и вызывается всякий раз, когда изменяется состояние процесса в результате сообщения или прерывания .

В отличие от процессов, выполнение самого микроядра никогда не планируется. Вход в микроядро – прямой результат вызовов его или процессом, или аппаратным прерыванием.

Системные процессы

Все услуги QNX, кроме обеспечиваемых микроядром, поддерживаются стандартными процессами QNX. Типичная конфигурация QNX имеет следующие системные процессы:

- менеджер процессов (Proc);
- менеджер файловой системы (Fsys);
- менеджер устройств (Dev);
- менеджер сети (Net).

Системные процессы и пользовательские процессы

Системные процессы фактически не отличаются от любой пользовательской программы – они не имеют никаких недоступных процессам пользователя частных или скрытых интерфейсов.

Именно эта архитектура обеспечивает QNX беспрецедентную расширяемость. Так как большинство услуг операционной системы обеспечивают стандартные процессы QNX, то расширить операционную систему очень просто: вы только добавляете новые программы.

Фактически, граница между операционной системой и приложением может быть очень размытой. Единственное реальное различие между системными услугами и приложениями – то, что эти услуги распределяют ресурсы для клиентов.

Предположим, что вы написали сервер базы данных. Как такой процесс должен быть классифицирован?

Так же, как файловая система допускает запросы (в QNX-сообщения) к открытым файлам и считывает/записывает данные, работает и сервер базы данных. Они очень похожи (несмотря на то, что запросы серверу базы данных могут быть более сложными) предоставляемыми наборами примитивов (основанными на сообщениях), обеспечивающими в свою очередь доступ к ресурсу. Это независимые процессы, которые могут быть написаны конечным пользователем и запущены при необходимости.

Сервер базы данных мог бы считаться системным процессом при одних установках и приложением – при других. Это действительно не имеет значения.

Важно то, что QNX позволяет реализовывать такие процессы без модификации типовых элементов операционной системы.

Драйверы устройств

Драйверы устройств – процессы, которые избавляют операционную систему от необходимости иметь дело со всеми деталями, требуемыми для поддержки определенных аппаратных средств.

Поскольку драйверы запускаются как стандартные процессы, добавление нового драйвера к QNX не затрагивает любую другую часть операционной системы. Единственное изменение, которое нужно среде QNX – запуск нового драйвера.

Завершив инициализацию, драйверы могут:

- исчезать как стандартные процессы, становясь расширениями системного процесса, с которым они связаны;
- сохранять индивидуальность как стандартные процессы.

Межпроцессное взаимодействие

Когда несколько процессов выполняется одновременно, например, как в типичных многозадачных средах в реальном масштабе времени, операционная система должна обеспечить механизмы для связи процессов друг с другом.

Межпроцессное взаимодействие (IPC) – ключ к проектированию приложения как набора взаимодействующих процессов, где каждый процесс выполняет свою четко определенную задачу.

QNX обеспечивает простой, но мощный набор средств межпроцессного взаимодействия, которые очень упрощают работу по созданию приложений, состоящих из взаимодействующих процессов.

QNX как операционная система с механизмом передачи сообщений

QNX была первой коммерческой операционной системой такого рода, использовавшей передачу сообщений как фундаментальное средство межпроцессного взаимодействия. QNX обязана многим из своей мощности, простоты и элегантности законченной интеграции метода передачи сообщений во всю систему.

Сообщение в QNX – пакет байтов, передающийся от одного процесса к другому. QNX не придает никакого значения содержанию сообщения – данные в сообщении имеют значение только для отправителя сообщения и для его получателя.

Передача сообщений позволяет процессам не только передавать данные друг другу, но также и обеспечивает средства синхронизации выполнения

нескольких процессов. Когда процессы посылают, принимают и отвечают на сообщения, они подвергаются различным «изменениям состояния», которые влияют на то, когда, и как долго, они могут выполняться. Зная их состояния и приоритеты, микроядро может распланировать все процессы настолько эффективно, насколько возможно использовать доступные ресурсы центрального процессора.

Приложения в реальном времени и другие критические приложения вообще требуют надежной формы связи между процессами, потому что процессы, которые составляют такие приложения, строго взаимосвязаны. Предлагаемая QNX передача сообщений обеспечивает упорядоченность и надежность.

Сеть QNX

В самой простой форме локальная сеть обеспечивает механизм для совместного использования файлов и периферийных устройств несколькими соединенными компьютерами. QNX выходит далеко за рамки этой простой концепции и интегрирует сеть в единый гомогенный набор ресурсов.

Любой процесс на любой машине в сети может непосредственно использовать любой ресурс на любой другой машине. С точки зрения приложения нет никакого различия между локальным или удаленным ресурсом – доступ к ним обеспечивается обычными средствами. Если бы это было не так, программа нуждалась бы в специальном коде, чтобы иметь возможность сообщить, находится ли ресурс типа файла или устройства на локальном компьютере или на некотором другом узле сети.

Пользователи могут обращаться к файлам где-нибудь в сети, пользоваться любыми периферийными устройствами и выполнять приложения на любой машине сети (если они имеют соответствующие полномочия). Процессы по всей сети могут связываться тем же самым способом, что и процессы, выполняющиеся на одной и той же машине. Вездесущая передача сообщений QNX обеспечивает мобильную организацию сети.

QNX разработана как сетевая операционная система. Сеть QNX похожа скорее на mainframe, чем на набор микроЭВМ. Пользователи просто осведомлены о большом наборе ресурсов, доступных для использования любому приложению. Но, в отличие от mainframe, QNX обеспечивает высокочувствительную среду, так как соответствующая вычислительная мощность для удовлетворения потребностей пользователя может быть доступна на любом узле.

В среде управления производственными процессами, например, PLC и другие устройства ввода-вывода реального времени могут требовать больше ресурсов, чем иные, менее критические приложения типа текстового процессора. Сеть QNX достаточно чутко реагирует для того, чтобы поддержать

оба вида приложений в одно и то же время – QNX позволяет сосредоточить вычислительную мощность там, где и когда это необходимо.

Сети QNX могут быть объединены с помощью различных аппаратных средств и стандартных промышленных протоколов. Новая архитектура сети может быть представлена в любое время без помех для операционной системы, так как она полностью прозрачна для прикладных программ и пользователей.

Каждому узлу в сети QNX назначается уникальный номер, который становится его идентификатором. Этот номер – единственное средство, позволяющее определить, работает ли QNX как сеть или как операционная система одиночного компьютера.

Эта степень прозрачности – еще один пример выдающейся мощности архитектуры передачи сообщений QNX. Во многих системах важные функции вроде организации сети, межпроцессного взаимодействия или даже передачи сообщений сформированы на вершине операционной системы, а не интегрированы непосредственно в ядро. Результат – часто неуклюжий, неэффективный интерфейс с «двойными стандартами».

QNX основана на том принципе, что эффективная связь – ключ к эффективной работе. Таким образом, передача сообщений – краеугольный камень архитектуры QNX, увеличивающий эффективность всех транзакций для всех процессов во всей системе, вне зависимости от того, сообщаются ли они через системную шину компьютера или через километр кабеля.

ОС РВ VxWorks

VxWorks – исполнительная (executive) операционная система реального времени (ОСРВ) на базе микроядра, разрабатываемая компанией Wind River Systems.

Название *VxWorks* получилось из игры слов с названием ОС РВ VRTX, созданной компанией Ready Systems. В начале восьмидесятых VRTX была достаточно новым и сырым продуктом и ее нельзя было использовать как полноценную операционную систему. Компания Wind River приобрела права на распространение расширенной версии VRTX под названием VxWorks. Доработки и расширения, внесённые компанией WindRiver, позволили создать систему, которая работала (например, VXWorks имела файловую систему и интегрированную среду разработки) таким образом. Название VxWorks может означать VRTX now Works («VRTX теперь работает») или VRTX that Works}} («VRTX, которая работает»).

Когда стало ясно, что Ready Systems может разорвать контракт на распространение VRTX, в Wind River было разработано собственное ядро операционной системы, которое заменило VRTX. Базовая функциональность нового ядра VxWorks была такой же как у VRTX.

Операционные системы реального времени семейства VxWorks корпорации WindRiver Systems предназначены для разработки программного обеспечения (ПО) встраиваемых компьютеров, работающих в системах жесткого реального времени. Операционная система VxWorks обладает кросс-средствами разработки программного обеспечения (ПО), т.е. разработка ведется на инструментальном компьютере (host) в среде Tornado для дальнейшего ее использования на целевом компьютере (target) под управлением системы VxWorks.

Операционная система VxWorks построена на базе микроядра, т.е. на самом нижнем непрерываемом уровне ядра (WIND Microkernel) обрабатываются только планирование задач и управление их взаимодействием/синхронизацией. Вся остальная функциональность операционного ядра – управление памятью, вводом/выводом и пр. – обеспечивается на более высоком уровне и реализуется через процессы. Это обеспечивает быстродействие и детерминированность ядра, а также масштабируемость системы.

VxWorks может быть скомпонована как для небольших встраиваемых систем с жесткими ограничениями для памяти, так и для сложных систем с развитой функциональностью. Более того, отдельные модули сами являются масштабируемыми. Конкретные функции можно убрать при сборке, а специфические ядерные объекты синхронизации можно опустить, если приложение в них не нуждается.

Хотя система VxWorks является конфигурируемой, т.е. отдельные модули можно загружать статически или динамически, нельзя сказать, что в ней используется подход, основанный на компонентах. Все модули построены над базовым ядром и спроектированы таким образом, что не могут использоваться в других средах.

Ядро VxWorks обладает следующими параметрами:

- количество задач не ограничено;
- число уровней приоритетов задач – 256'
- планирование задач возможно двумя способами – вытеснение по приоритетам и циклическое;
- средствами взаимодействия задач служат очереди сообщений, семафоры, события и каналы (для взаимодействия задач внутри ЦП), сокеты и удаленные вызовы процедур (для сетевого взаимодействия), сигналы (для управления исключительными ситуациями) и разделяемая память (для разделения данных);
- для управления критическими системными ресурсами обеспечивается несколько типов семафоров: двоичные, счетные (counting) и с приоритетным наследованием;

- поддерживается детерминированное переключение контекста.

В VxWorks обеспечивается как основанный на POSIX, так и собственные алгоритмы планирования (wind scheduling). Оба варианта включают вытесняющее и циклическое планирование. Различие между ними состоит в том, что wind scheduling применяется на системном базисе, в то время как алгоритмы POSIX-планирования используются на базисе процесс-за-процессом.

В VxWorks все задачи системы и приложений разделяют единственное адресное пространство, что чревато нарушением стабильности системы из-за неисправности какого-либо приложения. Необязательный компонент VxVMI дает возможность каждому процессу иметь свою собственную виртуальную память, т.е. реализует механизм защиты памяти.

Чтобы достичь быстрой обработки внешних прерываний, программы обработки прерываний в VxWorks выполняются в специальном контексте вне контекстов потоков, что позволяет выиграть время, которое обычно тратится на переключение контекстов. Следует отметить, что Си-функция, которую пользователь присоединяет к вектору прерывания, на самом деле не является фактической ISR. Прерывания не могут непосредственно обращаться к С-функциям. Адрес ISR запоминается в таблице векторов прерываний, которая вызывается аппаратно. ISR выполняет некую начальную обработку (сохранение регистров и подготовку стека), а затем вызывается С-функция, которая была присоединена пользователем.

VSPWorks – это весьма популярная и достаточно мощная ОС на основе VxWorks. VSPWorks спроектирована специально для систем, основанных на DSP. Она обеспечивает многозадачный режим с приоритетами и поддержку быстрых прерываний на процессорах DSP и ASIC. ОС РВ VSPWorks следует модели единственного виртуального процессора, что значительно упрощает распределение приложений в многопроцессорной системе, сохраняя при этом производительность жесткого реального времени. VSPWorks является модульной и масштабируемой.

ОСРВ VSPWorks обладает многослойной структурой, что служит хорошей основой для абстрагирования и переносимости. Центром системы служит сильно оптимизированное наноядро (nanokernel), которое способно управлять совокупностью процессов. Ниже наноядра находятся программы, обслуживающие прерывания, выше наноядра располагается микроядро, которое управляет многозадачным режимом с приоритетами С/С++ задач.

Примеры использования

- Контроллеры фирмы National Instruments, такие как CompactRio, Compact FieldPoint и Compact Vision System.

- Нобот ASIMO фирмы Honda.
- Косические аппараты для исследования планеты Марс Spirit и Opportunity.
- Многофункциональная автоматическая межпланетная станция НАСА Mars Reconnaissance Orbiter.
- Аппарат НАСА, предназначенный для изучения Марса Phoenix Mars Lander.
- Космический зонд НАСА Deep Impact space probe.
- Космический аппарат Stardust.
- Аэробус Boeing 787.
- Аэробус Boeing 747-8.
- Принтер Xerox Phaser и другие, основанные на Adobe PostScript принтеры.
- Экспериментальная промышленная и физическая система управления (Experimental Physics and Industrial Control System, EPICS).
- Процессоры для обработки графики DIGIC II и DIGIC III для фотоаппаратов фирмы Canon.
- Телекоммуникационный спутник Thuraya SO-2510 и ThurayaModule.
- Вертолет Apache Longbow.
- Радиолокационная система ALR-67(V)3 Radar Warning Receiver используемая в тактическом истребителе F/A-18E/F Super Hornet.
- Космический телескоп James Webb.
- Автомобильные системы Siemens VDO, автомобильные навигационные системы используемые фирмами BMW, Volkswagen, Opel и др.;
- Внешние RAID контроллеры фирмы LSI Corporation, используемые фирмами IBM, Silicon Graphics, Sun Microsystems, Teradata, Dell, Sepaton, BlueArc и др.

ОС РВ ОС2000

ОС2000 (ОС РВ «Багет 2.0» – микроядерная исполнительская операционная система реального времени (ОС РВ) для процессоров Intel и MIPS (Л1876ВМ1, аналог R3000 российского производства, завод Ангстрем). Разработана Научно-исследовательским институтом системных исследований при Российской Академии Наук (НИИСИ РАН), сертифицирована министерством обороны РФ для использования в военной вычислительной технике. Поставляется вместе с компьютерами серии Багет. ОС 2000 считается

первой серьезной отечественной операционной системой для встроенных систем. Интерфейс и дизайн системы похож на ОС РВ VxWorks компании Wind River System. Есть мнение, что реализация ОС2000 является полностью российской, существует только совместимость по системным вызовам.

Операционная система ОС2000 была разработана в 1998-2001 гг. отделом системного программирования НИИСИ РАН. К настоящему моменту выпущено три издания ОС2000. В качестве основы для пользовательского интерфейса был выбран стандарт POSIX 1003.1 1996 г. (были реализованы все функции за исключением управления процессами). Взаимодействие по сети (аппарат сокетов) было реализовано в соответствии со стандартом POSIX 1003.1 2001-2004 гг.

В качестве основного языка программирования ОС2000 предполагается использовать язык С.

Разработка операционной системы реального времени ос2000 базируется на следующих основополагающих принципах:

- соответствие международным стандартам;
- мобильность;
- использование концепции микроядра;
- использование объектно-ориентированного подхода;
- использование свободно распространяемого программного обеспечения;
- распространение ОС вместе со средствами разработки прикладных программ.

При разработке ОС2000 использовались следующие международные стандарты:

- POSIX 1003.1, стандарт на мобильные операционные системы (программный интерфейс);
- стандарт С, описывающий язык и библиотеки языка С.

Изначально POSIX разрабатывался с целью устранить разнобой в различных UNIX-системах и тем самым способствовать мобильности прикладных программ. UNIX был ориентирован не на решение задач реального времени, а на обеспечение одновременного доступа к ЭВМ со стороны нескольких пользователей, которые слабо взаимодействуют друг с другом или вообще не взаимодействуют.

Редакция POSIX 1996 года уже охватывает (в качестве необязательной части) основные функции операционных систем реального времени:

- потоки управления (threads);
- сигналы реального времени;
- средства синхронизации (семафоры, мьютексы и условные переменные);
- очереди сообщений;

- высокоточные таймеры;
- асинхронный ввод/вывод.

Стандарт POSIX продолжает развиваться, в том числе и в части, касающейся операционных систем реального времени. В настоящее время разрабатывается стандарт POSIX 1003.1j, также ориентированный на системы реального времени.

Операционная система ОС2000 полностью соответствует стандарту POSIX в части, относящейся к реальному времени. Те части стандарта, которые не относятся к системам реального времени (традиционный UNIX) не реализованы.

Для организации псевдо-параллельной обработки в POSIX используются два понятия – процессы и потоки управления. Процессы в POSIX являются держателями ресурсов (память, таблица открытых файлов и др.) и работают в значительной степени независимо друг от друга. Одной из функций ОС является защита процессов от нежелательного воздействия друг на друга.

В рамках процесса может выполняться один или несколько потоков управления. Потоки управления, выполняемые в рамках одного процесса, совместно используют его ресурсы. Для систем реального времени типичным является совместное использование ресурсов, поэтому в ОС2000 реализованы только потоки управления, но не процессы (с точки зрения POSIX в системе имеется только один процесс).

В рамках стандарта С реализованы математические функции (sin, exp, log и др.), функции обработки символов и строк, функции распределения памяти и др. Эти функции хорошо знакомы всем тем, кто разрабатывает программы на языке С.

К сожалению, стандарты не описывают все интерфейсы современных ОС реального времени. Это относится, в частности, к сетевым средствам и к графике. С другой стороны, некоторые свободно распространяемые программные продукты стали стандартами де-факто вследствие их широкого распространения. В силу этого, в ОС2000 использован аппарат сокетов операционной системы FreeBSD и графический интерфейс X Window.

Для эмуляции протокола Ethernet в многопроцессорных системах, использующих шину VME, будет использован стандарт ANSI/VITA. Этот стандарт позволяет взаимодействовать через шину VME и общую память различным как по аппаратуре, так и по используемой операционной системе процессорным платам.

Использование стандарта при разработке операционной системы выгодно как производителю операционной системы, так и ее пользователям. Если операционная система имеет уникальный интерфейс, то перенос прикладной программы с одной операционной системы на другую является довольно

трудоемким делом. В этом случае пользователь ОС попадает в зависимость от ее разработчика. Эта зависимость особенно ощутима, если нет версии ОС для аппаратной платформы, на которую нужно перенести прикладную программу. Перенос прикладных программ с одной ОС на другую значительно проще, если обе они удовлетворяют стандартам.

С целью повышения мобильности операционная система разбита на три части:

- не зависящая от аппаратуры;
- зависящая только от типа центрального процессора;
- пакет поддержки модуля (платы) (BSP).

Не зависящая от аппаратуры часть ОС имеет самый большой объем и написана полностью на языке С. Перенос этой части на другие платформы несложен.

Та часть ОС, которая зависит только от типа процессора, написана на языке С или на Ассемблере и имеет сравнительно небольшой объем. Туда входят, например, функции запоминания и восстановления контекста, пролог и эпилог диспетчера прерываний.

Пакет поддержки модуля (ППМ) содержит ту часть ОС, которая зависит от конкретной ЭВМ (платы). ППМ, в частности, содержит драйверы устройств и диспетчер прерываний (за исключением пролога и эпилога).

Отметим, что граница между этими частями не является жесткой. Например, некоторые не зависящие от аппаратуры функции, могут быть переписаны с использованием Ассемблера с целью повышения скорости. В этом случае они станут зависимыми от типа процессора.

ППМ и, в частности, драйверы поставляются вместе с исходными текстами и могут быть изменены пользователем. Внесение изменений в драйверы, а также разработка новых драйверов и включение их в операционную систему производится путем внесения изменений в исходные тексты ППМ. При этом не нужно вносить изменения в ядро операционной системы.

В настоящее время ОС2000 содержит пакет поддержки модуля для ЭВМ серии «Багет» с процессором 1В578 (совместимого с процессором MIPS R3000) и для PC-совместимых компьютеров (с процессором Intel).

Архитектура программного обеспечения

При использовании ОС2000 программное обеспечение системы реального времени состоит из операционной системы и прикладной программы. В системах реального времени граница между операционной системой и прикладной программой не так резко очерчена, как в случае традиционных операционных систем. В частности, в системах реального времени прикладная

программа может непосредственно вызывать те функции, которые в случае традиционных ОС могут выполняться только операционной системой (например, за прет или разрешение прерываний). Такая «свобода» позволяет повысить эффективность системы реального времени, но накладывает большую ответственность на прикладного программиста.

Для процессоров с архитектурой MIPS прикладная программа выполняется в режиме ядра и виртуальная память не применяется. Это позволяет заметно сократить время обращения к функциям операционной системы, сократить время переключения контекста и увеличить скорость выполнения прикладных программ и сделать ее предсказуемой.

Прикладная программа представляет собой совокупность потоков управления (пользовательских потоков управления). При инициализации системы порождается корневой поток управления прикладной программы, при необходимости другие потоки управления прикладная программа порождает динамически.

Операционная система состоит из ядра и системных потоков управления. Ядро выполняет функции планирования, синхронизации и взаимодействия потоков управления, а также низкоуровневые операции ввода/вывода. Функции ядра выполняются в контексте вызвавшего его потока управления или функции обработки прерывания. Микроядро представляет собой небольшую часть ядра ОС, функциями которой пользуются другие части ОС. Микроядро содержит функции управления потоками нижнего уровня (включая планировщик) и быстрые средства синхронизации (взаимное исключение|взаимного исключения). Все другие функции (например, захват и освобождение семафора, низкоуровневые операции ввода/вывода) выполняются вне микроядра, используя его функции.

Системные потоки выполняют более сложные функции операционной системы, такие как ввод/вывод информации по сети или обмен информацией с файловой системой. Использование системных потоков для сложных и протяженных во времени функций ОС позволяет продолжать работу параллельно с выполнением этих функций. В рамках таких потоков можно выполнять часть функций, которые обычно выполняются в рамках обработчиков прерываний драйвера. Во-первых, такой подход дает определенные удобства при программировании, так как не все функции ОС, доступные в контексте потока, можно выполнять в контексте прерывания. Во-вторых, сокращается время выполнения функций обработки прерываний, что особенно важно для систем реального времени. Действительно, если низкоприоритетная задача ведет интенсивный ввод/вывод, то она будет мешать выполнению высокоприоритетной задачи, так как функция обработки прерывания драйвера более приоритетна, чем любой поток управления. Потери (времени) будут тем больше, чем больше времени будет занимать функция обработки прерываний.

Средства разработки

Для разработки прикладного программного обеспечения используется комплекс, состоящий из двух ЭВМ, соединенных по сети:

- инструментальная ЭВМ (компьютер с операционной системой типа UNIX (Linux));
- целевая ЭВМ (ЭВМ, для которой разрабатывается программное обеспечение).

Разработка программного обеспечения ведется на инструментальной ЭВМ. Средства разработки позволяют оттранслировать программу, написанную на языках С и Ассемблер, а также отлаживать программу, загруженную в целевую машину.

Потоки управления

При разработке программ бывает удобно разбить их на части (задачи), которые выполнялись бы одновременно (параллельно). В системах реального времени им могут соответствовать одновременно протекающие в реальном мире процессы. Если у ЭВМ только один процессор, то вычисления производятся не параллельно, а псевдо-параллельно. Процессор выделяется разным задачам попеременно в соответствии с принятой стратегией планирования.

Для организации псевдо-параллельной обработки в POSIX используются два понятия – процессы и потоки управления. Процессы в POSIX являются держателями ресурсов (память, таблица открытых файлов и др.) и работают в значительной степени независимо друг от друга. Одной из функций ОС является защита процессов от нежелательного воздействия друг на друга.

В рамках процесса может выполняться один или несколько потоков управления. Потоки управления, выполняемые в рамках одного процесса, совместно используют его ресурсы. Для систем реального времени типичным является совместное использование ресурсов, поэтому в ос2000 реализованы только потоки управления, но не процессы (с точки зрения POSIX в системе имеется только один процесс).

Взаимодействие с сетью

Программное обеспечение импортировано из UNIX BSD 4.2, что обеспечивает соответствие стандартам, используемым в Интернет.

Реализованы следующие протоколы семейства TCP/IP:

- ARP (Address Resolution Protocol);
- IP (Internet Protocol);
- ICMP (Internet Control Message Protocol);

- TCP (Transmission Control Protocol);
- UDP (User Datagram Protocol);
- FTP (File Transfer Protocol);
- TELNET(Terminal emulation).

Пользовательский интерфейс (API) совпадает с UNIX BSD 4.2. Программное обеспечение портировано из UNIX BSD 4.2

Использование

- Бортовые компьютеры семейства Багет для автоматизированных систем управления войсками и оружием, в подвижных объектах (авиакосмическая техника, судостроение, сухопутная техника).
- Бортовой компьютер А-15АР для использования в оперативно-тактических комплексах, авиационных и других объектах.

5. Инструментальные средства проектирования ПО ВВС

5.1. Компиляторы языков высокого уровня

Компилятор – транслятор, который осуществляет перевод всей исходной программы в эквивалентную ей результирующую программу в рамках такой же модели вычислений.

Транслятор – программа, которая принимает на вход программу на одном языке (он в этом случае называется исходный язык, а программа – исходный код), и преобразует её в программу, написанную на другом языке (соответственно, целевой язык и объектный код).

5.2. Язык программирования Си

Си (англ. C) – императивный язык программирования или язык, в основе которого лежит модель вычислений Фон-Неймана. Простота, легкость переноса компиляторов на разные аппаратные платформы, наличие указателей и битовых операций делает удобным использование языка Си для системного программирования, программирования встроенных систем и микроконтроллеров. Аскетичность языка Си с одной стороны, функциональность и мощь с другой, хорошо укладываются в принцип KISS.

Язык программирования Си разработан в начале 1970-х годов сотрудниками Bell Labs Кеном Томпсоном и Денисом Ритчи как развитие языка Би. Си был создан для использования в операционной системе (ОС) UNIX. С тех пор он был перенесен на многие другие операционные системы и стал одним из самых используемых языков программирования.

Си ценят за его эффективность; он является самым популярным языком для создания системного программного обеспечения. Его также часто используют для создания прикладных программ. Несмотря на то, что Си не разрабатывался для новичков, он активно используется для обучения программированию. В дальнейшем синтаксис языка Си стал основой для многих других языков.

Для языка Си характерны лаконичность, современный набор конструкций управления потоком выполнения, структур данных и обширный набор операций.

Проблемы языка Си

Многие элементы Си потенциально опасны, а последствия неправильного использования этих элементов зачастую непредсказуемы. Керниган говорит: «Си – инструмент, острый, как бритва: с его помощью можно создать и

элегантную программу, и кровавое месиво». В связи со сравнительно низким уровнем языка многие случаи неправильного использования опасных элементов не обнаруживаются и не могут быть обнаружены ни при компиляции, ни во время исполнения. Они часто приводит к непредсказуемому поведению программы. Иногда в результате неграмотного использования элементов языка появляются уязвимости в системе безопасности. Необходимо заметить, что использования многих таких элементов можно избежать.

Обращение к несуществующему элементу массива

Чаще всего источником ошибки является обращение к несуществующему элементу массива. Несмотря на то, что Си непосредственно поддерживает статические массивы, он не имеет средств проверки индексов массивов (проверки границ). Например, возможна запись в шестой элемент массива из пяти элементов, что, естественно, приведёт к непредсказуемым результатам. Частный случай такой ошибки называется ошибкой переполнения буфера. Ошибки такого рода приводят к большинству проблем с безопасностью.

Ошибки, связанные с указателями

Другим потенциальным источником опасных ситуаций служит механизм указателей. Указатель может указывать на абсолютно любой объект в памяти, включая даже и сам машинный код программы, что может приводить к непредсказуемым эффектам. Несмотря на то, что большинство указателей, как правило, указывают на безопасные места, они легко могут быть передвинуты в уже небезопасные области памяти с помощью арифметики указателей; память, на которую они указывают, может быть освобождена и использована по-другому («висячие указатели»); они могут быть не инициализированы («дикие указатели»); или же они просто могут получить любое значение путём приведения типов или присваивания значения другого повреждённого указателя. Другие языки пытаются решить эти проблемы путём использования более ограниченных типов – ссылок.

Неинициализированные автоматические переменные

Одна из таких проблем заключается в том, что автоматически и динамически создаваемые объекты не инициализируются, поэтому в начале они имеют такое значение, какое осталось в памяти, выделенной для них от ранее удалённых объектов. Такое значение полностью непредсказуемо, оно меняется от одной машины к другой, от запуска к запуску, от вызова функции к вызову. Если программа попытается использовать такое неинициализированное значение, то придёт к непредсказуемому результату. Большинство современных компиляторов пытаются обнаружить эту проблему в некоторых случаях.

Функции с переменным количеством элементов

Функции с переменным количеством аргументов также являются потенциальным источником проблем. В отличие от обычных функций, имеющих прототип, стандартом не регламентируется проверка функций с переменным числом аргументов. Если передаётся неправильный тип данных, то возникает непредсказуемый, если не фатальный результат. Например, семейство функций printf стандартной библиотеки языка Си, используемое для генерации форматированного текста для вывода, хорошо известно своим потенциально опасным интерфейсом с переменным числом аргументов, которые описываются строкой формата. Проверка типов в функциях с переменным числом аргументов является задачей каждой конкретной реализации такой функции, однако многие современные компиляторы проверяют типы в каждом вызове printf, генерируя предупреждения в случаях, когда список аргументов не соответствует строке формата. Следует заметить, что невозможно статически проконтролировать даже все вызовы функции printf, поскольку строка формата может создаваться в программе динамически, поэтому, как правило, никаких проверок других функций с переменным числом аргументов компилятором не производится. Для помощи программистам на Си в решении этих и многих других проблем было создано большое число отдельных от компиляторов инструментов. Такими инструментами являются программы дополнительной проверки исходного кода и поиска распространённых ошибок, а также библиотеки, предоставляющие дополнительные функции, не входящие в стандарт языка, такие как проверка границ массивов или ограниченная форма сборки мусора.

Проблемы стандартной библиотеки языка C

Ещё одной распространённой проблемой является то, что память не может быть использована снова, пока она не будет освобождена программистом с помощью функции free(). В результате программист может случайно забыть освободить эту память и продолжить её выделять, занимая всё большее и большее пространство. Это обозначается термином утечка памяти. Наоборот, возможно освободить память слишком рано, но продолжать её использовать. Из-за того, что система выделения может использовать освобождённую память по-другому, это ведёт к непредсказуемым последствиям. Эти проблемы решаются в языках со сборкой мусора. С другой стороны, если память выделяется в функции и должна освобождаться после выхода из функции, данная проблема решается с помощью автоматического вызова деструкторов в языке C++, или с помощью локальных массивов, используя расширения C99.

Оптимизация программ для микроконтроллеров

Программирование микроконтроллеров существенно отличается от программирования систем общего назначения, так как информационно-управляющие системы, встроенные системы и микроконтроллеры имеют

достаточно ограниченные ресурсы производительности, а также памяти для хранения программ и данных.

Простые правила оптимизации

- Разберитесь с опциями компилятора, отвечающими за оптимизацию. Опций может быть много. Для начала, вам достаточно знать, что оптимизировать можно размер исполняемого кода или быстродействие. Не забывайте, что при максимальной степени оптимизации ваша программа может начать работать не так, как вы планировали.
- Эпизодически рассматривайте листинг вашей программы на предмет эффективности (для этого, вам придется изучить ассемблер для данного микроконтроллера). Разные компиляторы выдают совершенно различный код, поэтому единых рецептов не может быть. Для каждого типа компилятора и для каждого микроконтроллера вам придется ставить эксперименты, в которых вам нужно будет понять по листингу, какие языковые конструкции порождают более оптимальный код. Зависимости могут быть разные, размер и быстродействие программы могут зависеть от положения унарных операторов -- и ++, использования указателей или массивов и так далее. В некоторых случаях вам нужно будет использовать глобальные переменные, в других – автоматические. Вам придется выбирать типы циклов и анализировать влияние разрядности переменных на код. В общем, все зависит от конкретной ситуации.
- Для того, чтобы массив с константами не копировался в ОЗУ, используйте ключевое слово **const**.

```
const BYTE letter[26] =
{0xEE,0x3E,0x9C,0x7A,0x9E,0x8E,0xBC,     // a-g
 0x6E,0x60,0x70,0x00,0x1C,0x00,0x2A,     // h-n
 0xFC,0xCE,0xFD,0x0A,0xB6,0x1E,0x7C,     // o-u
 0x00,0x00,0x00,0x76,0x00};              // v-z
```

- Не забывайте, что изменение архитектуры вашей программы может дать больший эффект, чем оптимизация в мелочах.
- При острой необходимости получения высокого быстродействия используйте ассемблерные вставки. Старайтесь не применять этот подход без крайней необходимости, так как ассемблерные вставки затрудняют перенос программы на другой процессор и усложняют ее понимание.

5.3. Коммерческие компиляторы

Основная масса инструментального обеспечения для разработки встраиваемых систем продается за деньги. Стоимость компиляторов или

наборов инструментальных средств составляет от сотен до нескольких тысяч евро.

Сравнительно небольшое количество производителей микропроцессоров предлагает свои компиляторы бесплатно (для примера AVR Studio фирмы Atmel и Softune Workbench фирмы Fujitsu).

Коммерческие компиляторы имеют следующие характерные особенности:

- официальная поддержка;
- хорошая документация;
- меньшее количество ошибок в генерируемом коде (к сожалению, не всегда и не у всех);
- улучшенная оптимизация кода;
- более дружелюбный интерфейс пользователя.

Как правило, начинающие разработчики гораздо проще разбираются в коммерческих инструментальных средствах.

5.4. Компиляторы языка Си для ВВС с лицензией GPL

Некоторое количество компиляторов, используемых во встроенных системах выпускается под лицензией GNU GPL. GNU General Public License (иногда переводят, как, например, Универсальная общественная лицензия GNU, Универсальная общедоступная лицензия GNU или Открытое лицензионное соглашение GNU) – лицензия на свободное программное обеспечение, созданная в рамках проекта GNU в 1988 г. Её также сокращённо называют GNU GPL, или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии (существует довольно много других лицензий, содержащих слова «general public license» в названии).

Цель GNU GpvL – предоставить пользователю права копировать, модифицировать и распространять (в том числе на коммерческой основе) программы (что по умолчанию запрещено законом об авторских правах), а также гарантировать, что и пользователи всех производных программ получат вышеперечисленные права. Принцип «наследования» прав называется «копилефт» (транслитерация английского «copyleft») и был придуман Ричардом Столлмэном. По контрасту с GPL, лицензии собственнического ПО очень редко дают пользователю такие права и обычно, наоборот, стремятся их ограничить, например, запрещая восстановление исходного кода.

Использование свободно распространяемых инструментальных средств порождает некоторое количество проблем.

- Существует необходимость в достаточно высоком начальным уровне подготовки разработчика.
- Свободно-распространяемые компиляторы генерируют менее

эффективный код.

- Официальная поддержка отсутствует (тем не менее, присутствует неплохая неофициальная).

В целом, инструментальные средства выпущенные в рамках лицензии GPL являются в некоторых случаях очень неплохой альтернативой коммерческим продуктам.

Компилятор GCC

GCC (GNU Compiler Collection) – набор компиляторов, предоставляемых проектом GNU. GCC является одним из компонентов набора инструментальных средств GNU Toolchain.

Работа над GCC начата Ричардом Столлманом в 1985 году. Затем GCC был переписан Леном Тауэром.

Кроме Си, в компиляторе GCC поддерживаются следующие языки программирования.

- Язык C++.
- Язык Ada.
- Язык Java.
- Язык Objective-C.
- Язык Objective-C++.
- Язык Fortran.

Официальный сайт GCC находится по адресу http://gcc.gnu.org/.

Компилятор SDCC

SDCC – Small Device C Compiler. Многоцелевой, оптимизирующий компилятор с языка Си (ANSI C) для микроконтроллеров на базе Intel MCS51, Maxim 80DS390, Zilog Z80, Motorola 68HC08. Весь исходный код компилятора доступен и распространяется под лицензией GNU GPL. SDCC использует ASXXXX и ASLINK, бесплатные, способные полностью перенастраиваться на новые платформы ассемблер и линкер. SDCC обладает расширенным набором команд, основанном на оригинальном наборе команд микроконтроллеров на базе Intel 8051, что позволяет ему более эффективно использовать аппаратные возможности конкретной платформы. Он также содержит дебаггер/симулятор уровня исходного кода и может формировать отладочную информацию для NoICE дебаггера.

Скачать последнюю версию компилятора можно по адресу http://sdcc.sourceforge.net/

5.5. Нестандартные расширения языка Си

Практически все компиляторы Си, предназначенные для микроконтроллеров имеют нестандартные расширения. К ним относятся:

- нестандартные типы данных (битовые переменные, регистры специального назначения (SFR);
- модификаторы для обращения к различным адресным пространствам;
- средства управления рентабельностью функций;
- ассемблерные вставки, способы взаимодействия между программой, написанной на ассемблере и на Си;
- критические секции, семафоры;
- абсолютная адресация;
- оверлеи.

К особенностям реализации относятся:

- способы оптимизации кода;
- модели памяти;
- способы передачи параметров и получения результатов в функциях.

Расширения языка Си для компилятора GCC

GNU C обеспечивает некоторые языковые свойства, отсутствующие в стандарте ANSI C. (Опция `-pedantic` указывает GNU CC печатать предупреждающее сообщение, если какое-нибудь из этих свойств используется.) Чтобы проверить доступность этих свойств в условной компиляции, посмотрите предопределенный макрос __GNUC__, который всегда определен под GNU CC.

Эти расширения доступны в C и в Objective C. Большая часть из них также доступна в C++.

Составные операторы

Составной оператор, заключенный в скобки, может появляться в качестве выражения в GNU C. Это позволяет вам использовать циклы, операторы выбора и локальные переменные внутри выражения.

Напомним, что составной оператор – это последовательность операторов, заключенная в фигурные скобки; в этой конструкции скобки окружают фигурные скобки. Например:

```
({ int y = foo (); int z;
             if (y > 0) z = y;
             else z = - y;
             z; })
```

является правильным (хотя и несколько более сложным, чем необходимо) выражением для абсолютной величины foo(). Последним пунктом в составном операторе должно быть выражение, после которого следует точка с запятой; значение этого подвыражения служит значением всей конструкции. (Если вы используете какой-либо другой вид оператора последним внутри фигурных скобок, то тип конструкции – void, и таким образом она не имеет значения).

Локальные метки

Каждое выражение-оператор является областью, в которой могут быть объявлены локальные метки. Локальная метка – это просто идентификатор; вы можете делать переход на нее с помощью обычного оператора goto, но только изнутри выражения-оператора, к которому она принадлежит.

Объявление локальной метки выглядит так:

```
__label__ метка;
```

или

```
__label__ метка1, метка2, ...;
```

Объявления локальных меток должны идти в начале выражения-оператора сразу после `({` до любого обычного объявления. Объявление метки определяет имя метки, но не определяет саму метку. Вы должны сделать это обычным способом с помощью `метка:`, внутри выражения-оператора.

Получение адреса метки

Вы можете получить адрес метки, определенной в текущей функции (или объемлющей функции) с помощью унарной операции `&&`. Значение имеет тип void *. Это значение является константой и может быть использовано везде, где допускается константа этого типа. Например:

```
void *ptr;
 ...
 ptr = &&foo;
```

Чтобы использовать эти значения, вам нужно делать на них переход. Это делается с помощью вычисляемого оператора goto, `goto *выражение; `. Например:

```
goto *ptr;
```

Допустимо любое выражение типа void *.

Вложенные Функции

Вложенная функция – это функция, определенная внутри другой функции. Имя вложенной функции является локальным в блоке, где она определена. Например, здесь мы определяем вложенную функцию с именем square и вызываем ее дважды:

```
foo (double a, double b)
    {
        double square (double z) { return z * z; }

        return square (a) + square (b);
    }
```

Вложенная функция имеет доступ ко всем переменным объемлющей функции, которые видны в точке ее определения. Это называется «лексическая область действия». Например, рассмотрим вложенную функцию, которая использует наследуемую переменную с именем offset:

```
bar (int *array, int offset, int size)
    {
        int access (int *array, int index)
        {
            return array[index + offset];
        }

        int i;
        ...
        for (i = 0; i < size; i++)
        ... access (array, i) ...
    }
```

Определения вложенных функций разрешаются внутри функций, где допустимы определения переменных, то есть в любом блоке перед первым оператором в блоке.

Встроенные функции

Используя встроенные функции, описанные ниже, вы можете записать

полученные аргументы функции и вызвать другую функцию с теми же аргументами, не зная количество и типы аргументов.

Вы можете также записать возвращаемое значение этого вызова функции и позже вернуть это значение, не зная, какой тип данных функция пыталась вернуть (если вызывавшая функция ожидает этот тип данных).

__builtin_apply_args ()

Эта встроенная функция возвращает указатель типа void * на данные, описывающие, как выполнять вызов с теми же аргументами, которые были переданы текущей функции.

Функция сохраняет регистр указателя аргументов, адреса структурного значения и все регистры, которые могут быть использованы для передачи аргументов в функцию в блок памяти, выделяемый на стеке. Затем она возвращает адрес этого блока.

__builtin_apply (функция, аргументы, размер)

Эта встроенная функция вызывает 'функцию' (типа void (*)()) с копированием параметров, описываемых 'аргументами' (типа void *) и 'размером' (типа int).

Значение 'аргументы' должно быть значением, которое возвращено __builtin_apply_args (). Аргумент 'размер' указывает размер стековых данных параметров в байтах.

Эта функция возвращает указатель типа void * на данные, описывающие, как возвращать какое-либо значение, которое вернула 'функция'. Данные сохраняются в блоке памяти, выделенном на стеке.

Не всегда просто вычислить подходящее значение для 'размера'. Это значение используется __builtin_apply () для вычисления количества данных, которые должны быть положены на стек и скопированы из области входных аргументов.

__builtin_return (результат)

Эта встроенная функция возвращает значение, описанное 'результатом' из объемлющей функции. Вы должны указать в качестве 'результата' значение, возвращенное __builtin_apply ().

Именование типа выражения

Вы можете дать имя типу выражения, используя объявление typedef с инициализатором. Ниже показано, как определить имя как имя типа выражения:

```
typedef имя = выражение;
```

Это полезно в соединении с возможностью выражений-операторов. Ниже рассмотрим, как эти две возможности могут быть использованы, чтобы определить безопасный макрос «максимум», который оперирует с любым арифметическим типом:

```
#define max(a,b) \
({typedef _ta = (a), _tb = (b);  \
  _ta _a = (a); _tb _b = (b);     \
  _a > _b ? _a : _b; })
```

Смысл использования имен, которые начинаются с подчеркиваний для локальных переменных в том, чтобы избегать конфликтов с именами переменных, которые встречаются в выражениях, подставляющихся вместо a и b.

Ссылки на тип с помощью typeof

Другой способ сослаться на тип выражения – с помощью typeof. Синтаксис использования этого ключевого слова такой же, как и у sizeof, но семантически конструкция действует подобно имени типа, определенного с помощью typedef.

Есть два способа записи аргумента typeof: с выражением и с типом. Ниже показан пример с выражением:

```
typeof (x[0](1))
```

Здесь предполагается, что x является массивом функций; описанный тип является типом значений этих функций.

Ниже показан пример с именем типа в качестве аргумента:

```
typeof (int *)
```

Здесь описанный тип является типом указателей на int.

Если вы пишете заголовочный файл, который должен работать при включении в ANSI C программы, пишите __typeof__ вместо typeof.

Конструкция typeof может использоваться везде, где допустимо typedef-имя. Например, вы можете использовать ее в объявлении, в приведении или внутри sizeof или typeof.

Массивы нулевой длины

Массивы нулевой длины являются очень полезными в качестве последнего элемента структуры, который в действительности является заголовком объекта переменной длины:

```
struct line
{
     int length;
     char contents[0];
};

{
     struct line *thisline = (struct line *)
     malloc (sizeof (struct line) + this_length);
     thisline->length = this_length;
}
```

В стандартном C вы должны бы были дать contents длину 1, который означает, что вы либо должны терять память, либо усложнять аргумент malloc.

Массивы переменной длины

Автоматические массивы переменной длины допустимы в GNU C. Эти массивы объявляются подобно любым другим автоматическим массивам, но с длиной, которая не является константным выражением. Память выделяется в точке объявления и освобождается при выходе из блока. Например:

```
FILE *
          concat_fopen (char *s1, char *s2, char *mode)
          {
            char str[strlen (s1) + strlen (s2) + 1];
            strcpy (str, s1);
            strcat (str, s2);
            return fopen (str, mode);
          }
```

Переход вне области действия массива освобождает память. Переход в область действия недопустим.

Вы можете использовать функцию alloca, чтобы получить эффект во многом подобный массивам переменной длины. Функция alloca допустима во многих других реализациях C (но не во всех). С другой стороны, массивы переменной длины являются более элегантными.

Есть другие отличия между этими двумя методами. Место, выделяемое с помощью alloca, существует пока объемлющая функция не сделает возврат. Место для массива переменной длины освобождается, как только заканчивается область действия имени массива. (Если вы используете как массивы

переменной длины, так и alloca в одной и той же функции, освобождение массива переменной длины освободит все выделенное после с помощью alloca.)

Вы можете также использовать массивы переменной длины в качестве аргумента функции:

```
struct entry
          tester (int len, char data[len][len])
          {
            ...
          }
```

Длина массива вычисляется один раз при выделении памяти и вспоминается в области действия массива, если вы берете ее с помощью sizeof.

Если вы хотите передать массив первым, а длину после, вы можете использовать предварительное объявление в списке параметров – другое расширение GNU.

```
struct entry
          tester (int len; char data[len][len], int len)
          {
            ...
          }
```

'int len' перед точкой с запятой является предварительным объявлением параметра и служит тому, чтобы сделать имя len известным при разборе объявления data.

Вы можете писать любое число таких предварительных объявлений параметров в списке параметров. Они могут разделяться запятыми или точками с запятыми, но последнее из них должно кончаться точкой с запятой, за которой следуют «реальные» объявления параметров. Каждое предварительное объявление должно соответствовать «реальному» объявлению в имени параметра и типе данных.

Макросы с переменным числом аргументов

В GNU C макрос может получать переменное число аргументов, во многом подобное функции. Синтаксис определения макроса очень похож на используемый для функций. Пример:

```
#define eprintf(format, args...)  \
           fprintf (stderr, format , ## args)
```

Здесь args – это остаточный аргумент, он принимает ноль или больше аргументов – столько, сколько содержит вызов. Все они вместе с запятыми

между ними образуют значение args, которое подставляется в тело макроса там, где используется args. Таким образом, мы имеем следующее расширение:

```
eprintf ("%s:%d: ", input_file_name, line_number)
==>
fprintf (stderr, "%s:%d: " , input_file_name, line_number)
```

Заметим, что запятая после строковой константы идет из определения eprintf, в то время как последняя запятая идет из значения args.

Смысл использования '##' в обработке случая, когда args не соответствует ни одного аргумента. В этом случае args имеет пустое значение. Тогда вторая запятая в определении становится помехой: если она прошла бы через расширение макроса, мы бы получили что-нибудь подобное:

```
fprintf (stderr, "success!\n" , )
```

что является неправильным синтаксисом C. '##' освобождает от запятой, поэтому мы получаем следующее:

```
fprintf (stderr, "success!\n")
```

Это специальное свойство препроцессора GNU C: '##' перед остаточным аргументом, который пуст, отбрасывает предшествующую последовательность непробельных символов из макроопределения.

Арифметика над указателями на void и на Функции

В GNU C поддерживаются операции сложения и вычитания с указателями на void и на функции. Это можно сделать, приняв размер void или функции равным 1.

Следствием этого является то, что операция sizeof также разрешается над void и над типами функций и возвращает 1.

Опция '-Wpointer-arith' требует предупреждения, если это расширение используется.

Неконстантные инициализаторы

Как в стандартном C++, элементы агрегатного инициализатора автоматической переменной не обязаны быть константными выражениями в GNU C. Ниже показан пример инициализатора с элементами, меняющимися во время выполнения:

```
  foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  ...
}
```

Выражения конструкторов

GNU C поддерживает выражения конструкторов. Конструктор выглядит как приведение, содержащее инициализатор. Его значение является объектом типа, указанного в приведении, содержащее элементы, указанные в инициализаторе.

Обычно указанный тип является структурой. Предположим, что struct foo и structure объявлены:

```
struct foo {int a; char b[2];} structure;
```

Ниже показан пример конструирования struct foo с помощью конструктора:

```
structure = ((struct foo) {x + y, 'a', 0});
```

Это эквивалентно написанному ниже:

```
{
        struct foo temp = {x + y, 'a', 0};
      structure = temp;
}
```

Вы можете также сконструировать массив. Если все элементы конструктора являются (или получаются из) простыми константными выражениями, подходящими для использования в инициализаторах, тогда конструктор является L-значением и может быть приведен к указателю на свой первый элемент, как показано ниже:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Конструкторы массива, чьи элементы не являются простыми константами, не очень полезны, потому что они не являются L-значениями.

Помеченные элементы в инициализаторах

Стандартный C требует, чтобы элементы инициализатора появлялись в фиксированном порядке, в том же самом, в котором элементы массива или структуры инициализируются.

В GNU C вы можете дать элементы в любом порядке, указывая индексы массива или имена полей структуры, к которым они применяются.

Чтобы указать индекс массива, напишите '[индекс]' или '[индекс] =' перед значением элемента. Например,

```
int a[6] = { [4] 29, [2] = 15 };
```

эквивалентно

```
int a[6] = { 0, 0, 15, 0, 29, 0 }
```

Значение индекса должно быть константным выражением, даже если инициализируемый массив является автоматическим.

Диапазоны Case

Вы можете указать диапазон последовательных значений в одной метке case так:

```
case LOW ... HIGH:
```

Будьте внимательны: пишите пробелы вокруг '...', в противном случае оно может быть разобрано неправильно.

Атрибуты функций

В GNU C вы можете объявить определенные вещи о функциях, вызываемых в вашей программе, которые помогают компилятору оптимизировать вызовы функций и более внимательно проверять ваш код.

Ключевое слово __attribute__ позволяет вам указывать специальные атрибуты при создании объявлений. За этим ключеным словом следует описание атрибута в двойных скобках. В данный момент для функций определены восемь атрибутов: noreturn, const, format, section, constructor, destructor, unused и weak. Другие атрибуты, включая section, поддерживаются для объявлений переменных.

Вы можете указывать атрибуты с '__', окружающими каждое ключевое слово. Это позволяет вам использовать их в заголовочных файлах, не заботясь о том, что могут быть макросы с тем же именем. Например, вы можете использовать __noreturn__ вместо noreturn.

Атрибут interrupt

Используйте этот атрибут для микропроцессоров ARM, AVR, CRX, M32C, M32R/D, m68k, MS1, и Xstormy16 для указания того, что данная функция является обработчиком прерывания. Компилятор автоматически сгенерирует код, необходимый для сохранения и восстановления контекста.

- Замечание 1: для процессоров семейства ARM внутри функции

прерывания будут разрешены.

- Замечание 2: для процессоров семейства ARM необходимо указать вид прерывания так, как это указано в примере:
-

```
void f () __attribute__ ((interrupt ("IRQ")));
```

Возможные значения параметра: IRQ, FIQ, SWI, ABORT и UNDEF.

В процессоре ARMv7-M тип прерывания игнорируется, и атрибут означает, что функцию можно вызывать с выровненным на границе слова указателем стека.

Ключевое слово noreturn

Несколько стандартных библиотечных функций, таких как abort и exit не могут вернуть управление. GNU CC знает это автоматически. Некоторые программы определяют свои собственные функции, которые никогда не возвращают управление. Вы можете объявить их noreturn, чтобы сообщить компилятору этот факт. Например:

```
void fatal () __attribute__ ((noreturn));

void
fatal (...)
{
... /* Печатает сообщение об ошибке. */ ...
  exit (1);
}
```

Ключевое слово *noreturn* указывает компилятору принять, что функция fatal не может возвратить управление. Тогда он может делать оптимизацию несмотря на то, что случится, если fatal вернет управление. Это делает код немного лучше. Более важно, что это помогает избегать ненужных предупреждений об инициализированных переменных.

Атрибут *noreturn* не реализован в GNU C версии ранее чем 2.5.

Ключевое слово const

Многие функции не используют никаких значений, кроме своих аргументов, и не имеют эффекта, кроме возвращаемого значения. Такая функция может быть объектом исключения общих подвыражений и оптимизации циклов аналогично арифметической операции. Такую функцию следует объявить с атрибутом const. Например,

```
int square (int) __attribute__ ((const))
```

говорит о том, что гипотетическую функцию square безопасно вызывать меньшее количество раз, чем сказано в программе.

Атрибут const не реализован в GNU C версии ранее 2.5.

Заметим, что функция, которая имеет параметром указатель и использует данные, на которые он указывает, не должна объявляться const. Аналогично, функция, которая вызывает не-const функцию, обычно не должна быть const.

Атрибут format

Format (тип, строка-индекс, первый-проверяемый).

Атрибут format указывает, что функция принимает аргументы в стиле printf или scanf, которые должны быть проверены на соответствие со строкой формата. Например, объявление

```
extern int my_printf (void *my_object, const char *my_format, ...)
__attribute__ ((format (printf, 2, 3)));
```

заставляет компилятор проверять параметры в вызове my_printf на соответствие printf-стилю строки формата my_format.

Параметр 'тип' определяет, как строка формата интерпретируется, и должен быть либо printf, либо scanf. Параметр 'строка-индекс' указывает, какой параметр является строкой формата (начиная с 1), а 'первый-проверяемый' является номером первого проверяемого аргумента. Для функций, у которых аргументы не могут быть проверены (таких как vprintf), укажите в качестве третьего параметра ноль. В этом случае компилятор только проверяет строку формата на корректность.

Компилятор всегда проверяет формат для функций ANSI библиотеки printf, fprintf, sprintf, scanf, vprintf, vfprintf, vsprintf, когда такие предупреждения запрашиваются (используя '-Wformat'), так что нет нужды модифицировать заголовочный файл 'stdio.h'.

Атрибут section

Section – «имя-секции».

Обычно компилятор помещает генерируемый код в секцию text. Однако иногда вам нужны дополнительные секции или, чтобы определенные функции оказались в специальных секциях. Атрибут section указывает, что функция живет в определенной секции. Например, объявление

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

помещает функцию foobar в секцию bar.

Атрибуты constructor и destructor

Атрибут constructor заставляет функцию вызываться автоматически перед выполнением main (). Аналогично, атрибут destructor заставляет функцию вызываться автоматически после того, как main () завершилась или вызвана exit (). Функции с этими атрибутами полезны для инициализации данных.

Атрибут unused

Этот атрибут, примененный к функции, означает, что функция, возможно, может быть неиспользуемой. GNU CC не будет порождать предупреждение для этой функции.

Атрибут weak

Атрибут weak приводит к тому, что объявление будет порождаться как слабый символ, а не глобальный. Это прежде всего полезно для определения библиотечных функций, которые могут быть переопределены пользовательским кодом, хотя это может быть использовано и с объявлениями не-функций.

Атрибут alias

Alias – «назначение».

Атрибут alias заставляет породить объявление как синоним другого символа, который должен быть указан. Например,

```
void __f () { /* делает что-либо */; }
void f () __attribute__ ((weak, alias ("__f")));
```

объявляет 'f' слабым синонимом для '__f'.

Указание атрибутов переменных

Ключевое слово __attribute__ позволяет вам указывать специальные атрибуты переменных или полей структуры. За этим ключевым словом следует спецификация атрибута в двойных скобках. Восемь атрибутов поддерживаются в данный момент для переменных: aligned, mode, nocommon, packed, section, transparent_union, unused, weak.

Вы можете указывать атрибуты с '__', окружающими каждое ключевое слово. Это позволяет вам использовать их в заголовочных файлах, не заботясь о том, что могут быть макросы с тем же именем. Например, вы можете использовать __aligned__ вместо aligned.

Атрибут aligned (выравнивание)

Этот атрибут определяет минимальное выравнивание для переменной или поля структуры, измеряемое в байтах. Например, объявление

```
int x __attribute__ ((aligned (16))) = 0;
```

заставляет компилятор размещать глобальную переменную x по 16-байтной границе. На 68040 это может быть использовано вместе с asm выражением, чтобы использовать инструкцию move16, которой требуются операнды, выравненные на 16 байт.

Вы можете также указать выравнивание полей структуры. Например, для создания пары int, выравненной на границу двойного слова, вы могли бы написать:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

Это является альтернативой созданию объединения с double членом, который заставляет выравнивать объединение на границу двойного слова.

Невозможно определять выравнивание функций, оно определяется требованиями машины и не может быть изменено. Вы не можете указать выравнивание для typedef имени, потому что такое имя является только синонимом, а не отдельным типом.

Как в предыдущих примерах, вы можете явно указать выравнивание (в байтах), которое вы хотели бы, чтобы использовал компилятор для данной переменной или поля структуры. В качестве альтернативы, вы можете оставить размер выравнивания и только попросить компилятор выравнивать переменную или поле по максимальному полезному выравниванию для целевой машины, для которой вы компилируете. Например, вы могли бы написать:

```
short array[3] __attribute__ ((aligned));
```

Атрибут aligned может только увеличить выравнивание, но вы можете уменьшить его с помощью указания packed. (см. ниже).

Заметим, что эффективность атрибутов aligned может быть ограничена ограничениями вашего линкера. Во многих системах линкер может только обрабатывать выравнивание переменных, не превышающее определенного предела. (Для некоторых линкеров максимальное поддерживаемое выравнивание может быть очень и очень малым.) См. документацию по вашему линкеру для дальнейшей информации.

Атрибут mode (вид)

Этот атрибут указывает тип данных для объявления – тип, который соответствует виду 'вид'. Это в действительности позволяет вам требовать целый или плавающий тип в соответствии с его размером. Вы можете также указать вид 'byte', чтобы указать вид, соответствующий однобайтовому целому, 'word' для вида однословного целого и 'pointer' для вида, используемого для

представления указателей.

Атрибут nocommon

Этот атрибут указывает GNU CC помещать переменную "общей", а выделять место для нее прямо.

Атрибут packed

Атрибут packed указывает, что переменная или поле структуры должно иметь минимальное возможное выравнивание – один байт для переменной и один бит для поля, если вы не указали большее значение с помощью атрибута aligned.

Ниже показана структура, в которой поле x запаковано так, что оно непосредственно следует за a:

```
struct foo
                  {
                    char a;
                    int x[2] __attribute__ ((packed));
                      };
```

Атрибут section ("имя-секции")

Обычно компилятор помещает объекты, которые он генерирует в секции типа data и bss. Однако иногда вам нужны дополнительные секции, или вам нужно, чтобы определенные переменные оказались в специальных секциях, например, чтобы отобразить специальное оборудование. Атрибут section указывает, что переменная (или функция) живет в определенной секции. Например, эта маленькая программа использует несколько особых имен секций:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data_copy __attribute__ ((section ("INITDATACOPY"))) = 0;

main()
{
    /* Инициализируем указатель стека */
    init_sp (stack + sizeof (stack));

    /* Инициализируем инициализированные данные */
    memcpy (&init_data_copy, &data, &edata - &data);

    /* Включаем последовательные порты */
    init_duart (&a);
    init_duart (&b);
}
```

Используйте атрибут section с инициализированным определением

глобальной переменной, как показано в примере. GNU CC выдает предупреждение и игнорирует атрибут section в неинициализированном объявлении переменной.

Атрибут transparent_union

Этот атрибут, примененный к переменной-аргументу функции, который является объединением, означает передавать аргумент таким же образом, каким передавался бы первый член объединения. Вы можете также использовать этот атрибут с typedef для типа данных объединения, затем он применяется ко всем аргументам функций с этим типом.

Атрибут unused

Этот атрибут, примененный к переменной, означает, что переменная, возможно, может быть неиспользуемой. GNU CC не будет порождать предупреждение для этой переменной.

Атрибут weak

Атрибут weak описан в разделе «Атрибуты Функций».

Для указания многочисленных атрибутов разделяйте их запятыми внутри двойных скобок. Например:
'__attribute__ ((aligned (16), packed))'.

Атрибуты для типов

Ключевое слово __attribute__ позволяет вам указывать специальные атрибуты struct и union типов при их определении. За этим ключевым словом следует спецификация атрибута в двойных скобках. Три атрибута поддерживаются в данный момент для типов: aligned, packed, transparent_union. Другие атрибуты допустимы для функций (см. Раздел [Атрибуты Функций]) и для переменных (см. Раздел [Атрибуты Переменных]).

Вы можете указывать атрибуты с '__', окружающими каждое ключевое слово. Это позволяет вам использовать их в заголовочных файлах, не заботясь о том, что могут быть макросы с тем же именем. Например, вы можете использовать __aligned__ вместо aligned.

Вы можете указывать атрибуты aligned и transparent_union либо в typedef объявлении, либо сразу после закрывающей скобки полного определения enum, struct или union типа, а атрибут packed – только после закрывающей скобки определения.

Атрибут aligned (выравнивание)

Этот атрибут определяет минимальное выравнивание (в байтах) для переменных указанного типа. Например, объявление

```
struct S { short f[3]; } __attribute__ ((aligned (8));
typedef int more_aligned_int __attribute__ ((aligned (8));
```

заставляет компилятор гарантировать, что каждая переменная, чей тип – struct S или more_aligned_int, будет размещаться и выравниваться на по меньшей мере 8-байтовой границе.

Заметим, что выравнивание любого данного struct или union типа, требуемое стандартом ANSI C, будет, по меньшей мере, максимальным выравниванием из выравниваний всех членов рассматриваемого struct или union типа.

Как в предыдущем примере, вы можете явно указать выравнивание (в байтах), которое вы хотите, чтобы использовал компилятор для данного типа. В качестве альтернативы, вы можете оставить размер выравнивания и попросить компилятор выравнивать тип по максимальному полезному выравниванию для целевой машины, для которой вы компилируете. Например, вы могли бы написать:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Атрибут aligned может только увеличить выравнивание, но вы можете уменьшить его с помощью указания packed (см. ниже).

Заметим, что эффективность атрибутов aligned может быть ограничена ограничениями вашего линкера. Во многих системах линкер может обрабатывать только выравнивание переменных, не превышающее определенного предела. (Для некоторых линкеров максимальное поддерживаемое выравнивание может быть очень и очень малым.) См. документацию по вашему линкеру для дальнейшей информации.

Атрибут packed

Этот атрибут, примененный к определению enum, struct или union типа, указывает, что для представления этого типа должно быть использовано минимальное количество памяти.

Указание этого атрибута для enum, struct или union типа эквивалентно указанию атрибута packed для каждого члена структуры или объединения. Указание флага '-fshort-enums' в командной строке эквивалентно указанию атрибута packed для всех описаний enum.

Вы можете указывать этот атрибут только после закрывающей скобки описания enum, но не в typedef объявлении.

Атрибут transparent_union

Этот атрибут, присоединенный к описанию типа union, показывает, что любая переменная этого типа при передаче функции должна передаваться также, как передавался бы первый член объединения. Например:

```
union foo
{
    char a;
    int x[2];
} __attribute__ ((transparent_union));
```

Для указания многочисленных атрибутов разделяйте их запятыми внутри двойных скобок. Например:

```
__attribute__ ((aligned (16), packed))
```

Расширения языка Си для компилятора Keil C51

Компилятор C51 фирмы Keil Software поддерживает стандарт ANSI и все аспекты программирования на C, описанные этим стандартом. Все дополнения к стандарту предназначены для поддержки контроллера 8051. Дополнения касаются следующих понятий:

- типы данных;
- типы памяти;
- модели памяти;
- указатели;
- реентерабельные функции;
- функции обработки прерываний;
- операционные системы реального времени;
- поддержка PL/M и A51.

Типы данных

Компилятор C51 поддерживает все типы данных, приведенные ниже в таблице. Скалярные переменные могут быть объединены в структуры, объединения и массивы. За некоторым исключением (что указано ниже), доступ к значениям переменных может быть получен с помощью указателей.

Таблица 1. Типы данных

Типы данных	Биты	Байты	Область значений
bit*	1		от 0 до 1
signed char	8	1	от -128 до +127
unsigned char	8	1	от 0 до 255
enum	16	2	от -32768 до +32767
signed short	16	2	от -32768 до +32767
unsigned short	16	2	от 0 до 65535
signed int	16	2	от -32768 до +32767
unsigned int	16	2	от 0 до 65535
signed long	32	4	от -2147483648 до 2147483647
unsigned long	32	4	от 0 до 4294967295
float	32	4	от +1.175494E-38 до +3.402823E+38
sbit*	1	0	от 0 до 1
sfr*	8	1	от 0 до 255
sfr16*	16	2	от 0 до 65535

**Типы bit, sbit, sfr и sfr16 являются специфичными для процессора 8051 и компилятора C51. Они не описаны стандартом ANSI, поэтому к ним нельзя обращаться через указатели.*

Типы данных sbit, sfr и sfr16 используются для обращения к специальным регистрам процессора 8051. Например, строка sfr P0 = 0x80; объявляет переменную P0 и назначает ей адрес специального регистра 0x80. Это адрес порта P0. Компилятор C51 производит автоматическое преобразование типов в случае, если результат относится к другому типу. Кроме того, можно выполнить принудительное приведение типов. В процессе приведения расширение знака к переменным знаковых типов добавляется автоматически.

Модификаторы памяти

Компилятор C51 поддерживает контроллеры типа 8051 и обеспечивает доступ ко всем областям памяти контроллера. Для каждой переменной можно точно указать область ее размещения в памяти.

Таблица 2. Модификаторы памяти

Модификаторы памяти	Описание
code	Память программ (64 Кб); выполняется следующий код: MOVC @A+DPTR.
data	Внутренняя память данных с прямой адресацией; самая быстрая работа с переменными (128 байт).
idata	Внутренняя память данных с косвенной адресацией; доступ ко всему адресному пространству (256 байт).
bdata	Бит-адресуемая внутренняя память данных; возможность обращаться как к битам, так и к байтам (16 байт).
xdata	Внешняя память данных (64 Kbytes); выполняется код: MOVX @DPTR.
pdata	Внешняя память данных со страничной организацией (256 байт); выполняется код: MOVX @Rn.

Обращение к внутренней памяти данных происходит гораздо быстрее, чем к внешней. Поэтому переменные, которые используются чаще других, следует размещать во внутренней памяти, а остальные – во внешней.

Применяя модификаторы памяти при объявлении переменной, можно указать, где именно она будет размещена.

Можно включить сведения о модели памяти в объявление переменной (точно так же, как атрибуты signed и unsigned), например:

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

Если в объявлении переменной модификатор памяти не указан, выбирается модель памяти, установленная по умолчанию. Аргументы функции и переменные класса памяти auto, которые не могут быть размещены в регистрах, также хранятся в области памяти, установленной по умолчанию.

Модель памяти, выбираемая в качестве модели по умолчанию, устанавливается с помощью директив компилятора SMALL, COMPACT и LARGE.

Модели памяти

Вы можете назначить модель памяти для тех аргументов функций,

автопеременных и переменных, в объявлении которых не указана соответствующая модель. Это делается с помощью директив **SMALL**, **COMPACT** и **LARGE**. Явное задание модели отменяет использование модели памяти по умолчанию.

Таблица 3. Модели памяти

Название	Описание
SMALL	В этой модели все переменные по умолчанию размещаются во внутренней памяти данных. Достигается тот же эффект, как при явном использовании модификатора data. При такой модели обращение к переменным оказывается очень эффективным. Но все объекты данных и стек должны размещаться во внутренней памяти. Размер стека имеет решающее значение, так как пространство, занимаемое стеком, зависит от глубины вложенности различных функций. Обычно, если компоновщик BL51 сконфигурирован для оверлейной загрузки переменных во внутреннюю память данных, лучше всего использовать малую модель памяти (small).
COMPACT	Если используется компактная модель памяти, все переменные по умолчанию загружаются во внешнюю память данных – точно так же, как при явном задании модификатора памяти pdata. В этой памяти может быть размещено до 256 байтов. Ограничения появляются вследствие использования косвенной адресации, когда обращение происходит через регистры R0 и R1. Данная модель памяти не так эффективна, как малая, и обращение к переменным происходит медленнее. Но компактная модель все же быстрее, чем большая. Старший бит адреса обычно устанавливается через порт 2, причем делается это вручную программистом.
LARGE	При использовании большой модели памяти все переменные по умолчанию размещаются во внешней памяти данных. Можно также явно указать модель памяти с помощью модификатора xdata. Для адресации используется указатель DPTR. Обращение к памяти через этот указатель не является эффективным, особенно если переменная имеет длину 2 или более байт. При использовании данной модели памяти код получается длиннее, чем при малой или компактной модели.

Примечание

Почти всегда следует использовать модель памяти SMALL: в этом случае вы получите самый быстрый, компактный и наиболее эффективный код. Однако у вас есть возможность явно указать нужную модель памяти. Модели, использующие внешнюю память, стоит использовать только в том случае, если

ваше приложение никаким образом не может работать при модели памяти SMALL.

Указатели

Компилятор C51 поддерживает объявление указателей с использованием символа звездочки (*). Указатели можно использовать для выполнения всех доступных в стандартном С операций.

Вследствие уникальности архитектуры контроллера 8051 и его производных, компилятор C51 поддерживает 2 вида указателей: память-зависимые и нетипизированные.

Нетипизированные указатели

Нетипизированные указатели объявляются точно так же, как указатели в стандартном С, например:

```
char *s; /* string ptr */

int *numptr; /* int ptr */

long *state; /* long ptr */
```

Нетипизированные указатели имеют размер 3 байта. В первом байте указывается модель памяти, во втором – старший байт смещения, в третьем – младший байт смещения.

Нетипизированные указатели используются для обращения к любым переменным, независимо от их размещения в памяти контроллера. Многие библиотечные функции используют эти указатели, при этом им совершенно неважно, в какой именно области памяти они размещаются.

Память-зависимые указатели

В объявления память-зависимых указателей всегда включается модификатор памяти. Обращение всегда происходит к указанной области памяти, например:

```
char data *str;/* ptr to string in data <nowiki>*/</nowiki>

int xdata *numtab;/* ptr to int(s) in xdata */

long code *powtab;/* ptr to long(s) in code */
```

Поскольку модель памяти определяется во время компиляции, типизированным указателям не нужен байт, в котором указывается модель.

Типизированные указатели могут иметь размер в 1 байт (указатели idata, data, bdata, и pdata) или в 2 байта (указатели code и xdata).

Сравнение память-зависимых и нетипизированных указателей

Можно существенно ускорить выполнение кода путем использования типизированных указателей. Приведенные ниже примеры программ показывают различия в размерах кода и данных, а также времени выполнения для двух видов указателей.

Таблица 4. Сравнение память-зависимых и нетипизированных указателей

Описание	**Указатель Idata**	**Указатель Xdata**	**Нетипизированный указатель**
Пример программы	char idata *ip; char val; val = *ip;	char xdata *xp; char val; val = *xp;	char *p; char val; val = *p;

Код, сгенерированный компилятором 8051	MOV R0,ip
	MOV val,@R0
		MOV DPL,xp +1
		MOV DPH,xp
		MOV A,@DPTR
		MOV val,A
			MOV R1,p + 2
			MOV R2,p + 1
			MOV R3,p
			CALL CLDPTR

Размер указателя	1 байт	2 байта	3 байта
Размер кода	4 байта	9 байт	11 байт кода + библиот.
Время выполнения	4 цикла	7 циклов	13 циклов

Реентерабельные функции

Реентерабельные функции могут быть вызваны одновременно разными процессами. При выполнении такой функции другой процесс может прервать ее выполнение и начать выполнять ту же функцию. В большинстве случаев функции C51 не могут быть вызваны рекурсивно или повторно до окончания выполнения. Причина этого в том, что аргументы функций и локальные переменные хранятся в строго определенных областях памяти. Однако атрибут reentrant позволяет объявить функцию как реентерабельную, которая может быть вызвана рекурсивно, например:

```
int calc (char i, int b) reentrant
{
int x;

    x = table [i];
    return (x * b);
}
```

Реентерабельные функции могут вызываться рекурсивно, а также *одновременно* двумя и более процессами. Подобные функции часто используются в приложениях, работающих в режиме реального времени, или в тех случаях, когда функция вызывается как из кода, так и по прерыванию.

Стек каждой реентерабельной функции размещается во внутренней или внешней памяти в зависимости от модели памяти.

Примечание: добавление атрибута reentrant при объявлении функции позволит сделать реентерабельными только те участки программ, которые могут быть вызваны повторно или рекурсивно. При этом вся программа не становится реентерабельной. Если вы объявите реентерабельной целую программу, она займет больше места в памяти.

Функции – обработчики прерываний

Компилятор C51 предоставляет возможность вызова функций при возникновении прерываний. Это дает возможность написания на языке С собственных обработчиков прерываний. Однако следует соблюдать осторожность в выборе номера прерывания и банка регистров. Компилятор автоматически генерирует вектор прерывания, а также код входа в обработчик и выхода из него. Атрибут interrupt, включенный в объявление функции, указывает на то, что данная функция обрабатывает прерывание. Кроме того, можно указать банк регистров для данного прерывания с помощью атрибута using.

```
unsigned int interruptcnt;
unsigned char second;


void timer0 (void) interrupt 1 using 2
{

    if (++interruptcnt == 4000)
    {

    second++;

    interruptcnt = 0;
    }
}
```

Передача параметров

Компилятор C51 размещает в регистрах до 3 аргументов функций. Это значительно повышает системную производительность, поскольку аргументы

не требуется сохранять в памяти и считывать оттуда. Передачей параметров можно управлять с помощью специальных директив – REGPARMS и NOREGPARMS.

Ниже приведен список регистров, используемых для размещения аргументов разных типов.

Таблица 5. Регистры, используемые для размещения аргументов разных типов

Количество аргументов	char, 1-байтный указатель	int, 2-байтный указатель	long, float	Нетипизированный указатель
1	R7	R6 & R7	R4 - R7	R1 - R3
2	R5	R4 & R5		
3	R3	R2 & R3		

Если в данный момент нет свободных регистров или аргументов слишком много, то для этих аргументов используются фиксированные области памяти.

Значения, возвращаемые функцией

Регистры микроконтроллера всегда используются для значений, возвращаемых функциями. Ниже приведена таблица типов возвращаемых данных и соответствующих им регистров.

Таблица 6. Типы возвращаемых данных и регистры

Тип возвращаемого значения	Регистры	Описание
bit	Флаг переноса	
char, unsigned char, 1-byte pointer	R7	
int, unsigned int, 2-byte pointer	R6 & R7	MSB в R6, LSB в R7
long, unsigned long	R4 - R7	MSB в R4, LSB в R7
float	R4 - R7	32-битный формат IEEE
generic pointer	R1 - R3	Тип памяти в R3, MSB R2, LSB, R1

Оптимальное выделение регистров

Компилятор C51 выделяет для регистровых переменных до 7 регистров процессора (в зависимости от контекста программы). Компилятору известны все регистры, содержимое которых изменилось в процессе выполнения функции. Компоновщик / загрузчик генерирует один файл на весь проект, содержащий информацию обо всех регистрах, измененных внешними функциями. В результате компилятор знает об изменениях в каждом регистре и может оптимально распределить регистры среди всех функций.

Добавление ассемблерного кода

К процедурам, написанным на ассемблере, можно легко обращаться из кода на языке С (и наоборот). Параметры функций передаются через регистры контроллера или, в случае использования директивы NOREGPARMS, путем загрузки их в фиксированные области памяти. Значения, возвращаемые функциями, всегда передаются через регистры.

Чтобы сгенерировать не объектный файл, а код, который может быть обработан ассемблером A51, нужно использовать директиву компилятора C51 SRC. Например, приведенный ниже текст на С:

```
unsigned int asmfunc1 (unsigned int arg)
{
    return (1 + arg);
}
```

Он откомпилирован с использованием директивы SRC и создает следующий код на ассемблере:

```
?PR?_asmfunc1?ASM1 SEGMENT CODE
PUBLIC _asmfunc1
RSEG ?PR?_asmfunc1?ASM1
USING 0
_asmfunc1:
;---- Variable 'arg?00' assigned to Register 'R6/R7' ----
MOV A,R7 ; load LSB of the int
ADD A,#01H ; add 1
MOV R7,A ; put it back into R7
CLR A
ADDC A,R6 ; add carry & R6
MOV R6,A
?C0001:
RET ; return result in R6/R7
```

Чтобы включить инструкции ассемблера в программу на С, следует использовать директивы препроцессора #pragma asm и #pragma endasm.

Расширения языка Си для компилятора SDCC

Модели памяти

В компиляторе языка Си SDCC для MCS51 существуют модели small, medium и large. В моделях medium и large все переменные без класса памяти попадают по умолчанию во внешнее ОЗУ (XRAM). В модели памяти small все переменные без класса памяти попадают во внутреннюю память данных.

Стек во внешней памяти

Опция *--xstack* позволяет располагать стек во внешней памяти, в сегменте pdata.

Абсолютная адресация

Переменным можно присваивать не только тип памяти (data, xdata, code), но и абсолютный адрес расположения.

```
__xdata __at (0x7ffe) unsigned int chksum;
```

В примере, приведенном выше, переменная *chksum* будет размещена по адресу 0x7ffe во внешней памяти XDATA. Необходимо заметить, что компилятор не резервирует места под переменные, определенные таким образом, и проверка на отсутствие пересечений с другими данными полностью лежит на программисте. Для проверки можно использовать файлы с расширениями .lst, .rst и .map.

Если вы произведете инициализацию памяти, линкер сможет обнаружить пересечение данных.

```
__code __at (0x7ff0) char Id[5] = ''SDCC'';
```

В случае использования устройств ввода-вывода, расположенных в адресном пространстве внешней памяти, необходимо использовать ключевое слово *volatile*, для того, чтобы оптимизатор компилятора не заменил обращение к устройству обращением к регистру общего назначения.

```
volatile __xdata __at (0x8000) unsigned char PORTA_8255;
```

В SDCC допускается указание абсолютного адреса расположения бита для битовых переменных.

```
__bit __at (0x02) bvar;
```

Использование абсолютной адресации битовой памяти может вызвать путаницу и оправдано только если вам хочется написать универсальную программу (см. пример ниже) для нескольких комплектов аппаратных средств.

```
extern volatile __bit MOSI;    /* master out, slave in */
extern volatile __bit MISO;    /* master in, slave out */
extern volatile __bit MCLK;    /* master clock */

    unsigned char spi_io(unsigned char out_byte)
    {
        unsigned char i=8;
        do {
            MOSI = out_byte & 0x80;
            out_byte <<= 1;
            MCLK = 1;
            if(MISO)
                out_byte += 1;
            MCLK = 0;
        } while(-i);
        return out_byte;
    }
```

Далее мы видим варианты определения битов для разных аппаратных средств.

```
// вариант 1
    __bit __at (0x80) MOSI;    /* I/O port 0, bit 0 */
    __bit __at (0x81) MISO;    /* I/O port 0, bit 1 */
    __bit __at (0x82) MCLK;    /* I/O port 0, bit 2 */
// вариант 2
    __bit __at (0x83) MOSI;    /* I/O port 0, bit 3 */
    __bit __at (0x91) MISO;    /* I/O port 1, bit 1 */
    __bit __at (0x92) MCLK;    /* I/O port 1, bit 2 */
```

Нестандартные типы данных

bit

Битовый тип данных. Возможные значения 0 и 1.

```
__bit test_bit;
```

sfr / sfr16 / sfr32 / sbit

Ключевые слова для определения регистров специального назначения. sbit – битовый регистр, sfr – 8 разрядов, sfr16 – 16 разрядов, sfr32 – 32 разряда. Эти ключевые слова используются для создания заголовочных файлов, позволяющих обращаться к регистрам специального назначения по именам.

```
__sfr __at (0x80) P0;  /* регистр специального назначения P0 по адресу 0x80
*/

    /* 16 разрядный регистр общего назначения для timer 0
       Старший байт значения находится по адресу 0x8C, младший по адресу 0x8A
*/
    __sfr16 __at (0x8C8A) TMR0;

    __sbit __at (0xd7) CY;  /* CY (Carry Flag, флаг переноса) */
```

Фрагмент заголовочного файла для ADuC 812

```
/*  Регистры с байтовым доступом */
__sfr __at ( 0x80 )  P0       ;
```

```
__sfr __at ( 0x81 )  SP       ;
__sfr __at ( 0x82 )  DPL      ;
__sfr __at ( 0x83 )  DPH      ;
__sfr __at ( 0x84 )  DPP      ;
...
__sfr __at ( 0xF2 )  ADCOFSH  ;
__sfr __at ( 0xF3 )  ADCGAINL ;
__sfr __at ( 0xF4 )  ADCGAINH ;
__sfr __at ( 0xF5 )  ADCCON3  ;
__sfr __at ( 0xF7 )  SPIDAT   ;
__sfr __at ( 0xF8 )  SPICON   ;
__sfr __at ( 0xF9 )  DAC0L    ;
...
/* Битовые регистры */
/* TCON */
__sbit __at ( 0x8F ) TF1      ;
__sbit __at ( 0x8E ) TR1      ;
__sbit __at ( 0x8D ) TF0      ;
__sbit __at ( 0x8C ) TR0      ;
__sbit __at ( 0x8B ) IE1      ;
...
/* Номера битов для байтовых регистров */
/* PCON bits */
#define IDL             0x01
#define PD              0x02
#define GF0             0x04
#define GF1             0x08
#define SMOD            0x80

...
/* Номер прерывания: адрес = (номер * 8) + 3  */
#define IE0_VECTOR      0   /* 0x03 Внешнее прерывание 0 */
#define TF0_VECTOR      1   /* 0x0b Таймер 0 */
#define IE1_VECTOR      2   /* 0x13 Внешнее прерывание 1 */
#define TF1_VECTOR      3   /* 0x1b Таймер 1 */
#define SI0_VECTOR      4   /* 0x23 Последовательный канал 0 */
```

Модификаторы памяти

```
/* указатель находится во внутренней памяти и ссылается на объект во внешней
памяти XDATA */

    __xdata unsigned char * __data p;

/* указатель находится во внешней памяти и ссылаетсмя на объект во внутренней
*/

    __data unsigned char * __xdata p;

/* указатель в пространстве памяти команд (CODE) ссылается на объект во
внешней памяти XDATA */

    __xdata unsigned char * __code p;

/* указатель в пространстве памяти команд (CODE) ссылается на объект в
пространстве памяти команд (CODE) */

    __code unsigned char * __code p;

/* указатель на функцию, находящийся во внутренней памяти  */

    char (* __data fp)(void);
```

Универсальный (generic) указатель в SDCC содержит дополнительную информацию о типе адресного пространства. В целях оптимизации объема кода лучше явно указывать тип адресного пространства.

```
/* универсальный указатель (generic) находится во внешней памяти XDATA */

    unsigned char * __xdata p;

/* универсальный указатель (generic) находится в пространстве памяти по
умолчанию */

    unsigned char * p;
```

Реентерабельность

Реентерабельность (Reentrant, повторное вхождение) – свойство императивной программы, позволяющее одновременное использование нескольких одинаковых программ в многозадачной среде. Для того, чтобы программа стала нереентерабельной, достаточно разместить переменные в глобальной области памяти в единственном экземпляре. Тогда несколько копий программ будет использовать один и тот же набор переменных, что приведет к непредсказуемым последствиям. Для обеспечения реентерабельности необходимо снабдить каждый экземпляр кода своим собственным объемом памяти для хранения данных. Как правило, для этих целей используют регистровые файлы и стек.

В зависимости от модели памяти и количества свободного места, автоматические переменные и параметры функции могут быть помещены в стек, или в пространство внешней или внутренней памяти. Последний вариант

делает функцию нереентерабельной. Для того, чтобы разместить автоматические переменные в стеке, можно воспользоваться опцией компилятора *--stack-auto* (или в тексте программы *#pragma stack-auto*), или ключевым словом *reentrant* при определении функции.

```
unsigned char foo(char i) __reentrant
    {
        ...
    }
```

Необходимо помнить, что в архитектуре MCS51 стек имеет очень небольшой объем. Поэтому опцией --stack-auto необходимо пользоваться экономно и с осторожностью. Для решения проблемы с размером стека можно явно указывать тип памяти для автоматической переменной и её абсолютный адрес.

```
unsigned char foo()
    {
        __xdata unsigned char i;
        __bit bvar;
        __data __at (0x31) unsigned char j;
        ...
    }
```

Оверлеи

Оверлей – механизм управления распределением ресурсов, при котором допускается их совместное использование. Как правило, под термином оверлей понимается механизм повторного использования блоков памяти данных и кода в вычислительных системах, работающих в рамках модели вычислений Фон-Неймана.

В нереентерабельных функциях одна и таже область памяти может быть использована повторно. SDCC использует оверлеи по умолчанию. Для отключения режима работы с оверлеями необходимо использовать *#pragma nooverlay*.

```
#pragma save
    #pragma nooverlay
    void set_error(unsigned char errcd)
    {
        P3 = errcd;
    }
    #pragma restore

    void some_isr () __interrupt (2)
    {
        ...
        set_error(10);
        ...
    }
```

В приведенном примере, использование *errcd* без *#pragma nooverlay* привело бы к непредсказуемым последствиям.

Обработчики прерываний

Обработчик прерывания в SDCC имеет следующий вид:

```
void timer_isr (void) __interrupt (1) __using (1)
    {
        ...
    }
```

Ключевое слово *__interrupt* определяет номер вектора прерываний, а слово *__using* – номер используемого регистрового банка. Явное указание номера регистрового банка позволяет уменьшить объем данных, сохраняемых в стеке при вызове обработчика. Предполагается, естественно, что этот регистровый банк кроме обработчика никто не будет использовать.

Если обработчик прерывания изменяет какие либо глобальные переменные, они должны быть определены с использованием ключевого слова *volatile*.

```
/* Номера обработчиков прерываний: адрес = (номер * 8) + 3 */

#define IE0_VECTOR      0       /* 0x03 external interrupt 0 */
#define TF0_VECTOR      1       /* 0x0b timer 0 */
#define IE1_VECTOR      2       /* 0x13 external interrupt 1 */
#define TF1_VECTOR      3       /* 0x1b timer 1 */
#define SI0_VECTOR      4       /* 0x23 serial port 0 */
```

Критические секции

Внутри критической секции SDCC генерирует код, который запрещает (в начале секции) и восстанавливает в исходное состояние (в конце) все прерывания. Необходимо помнить, что в большинстве случаев запрещать все

прерывания слишком накладно.

```
int foo () __critical
    {
        ...
        ...
    }
```

Ключевое слово __*critical* может использоваться совместно с ключевым словом *reentrant*.

Ключевое слово __*critical* может использоваться для защиты отдельных переменных.

```
__critical{ i++; }
```

Семафоры

Архитектура MCS51 позволяет проводить атомарные действия над битовыми переменными, что позволяет успешно реализовать простой бинарный семафор. SDCC генерирует такой код, если используется приведенный ниже шаблон исходного текста.

```
volatile bit resource_is_free;

    if (resource_is_free)
      {
        resource_is_free=0;
        ...
        resource_is_free=1;
      }
```

Пример использования бинарного семафора.

```
char x = 0;
volatile bit resource_is_free;

void sem( void )
{
    if (resource_is_free)
    {
        resource_is_free = 0;
        x = 10;
        resource_is_free = 1;
    }
}
```

Генерируемый SDCC код.

```
    jbc  _resource_is_free,00106$
    ret
00106$:
    mov  _x,#0x0A
    setb _resource_is_free
    ret
```

Ассемблерные вставки

Компилятор SDCC позволяет использовать ассемблерные вставки. Попробуем оптимизировать программу, представленную ниже.

```
unsigned char __far __at(0x7f00) buf[0x100];
    unsigned char head, tail;

/* if interrupts are involved see
section 3.8.1.1 about volatile */

    void to_buffer( unsigned char c )
    {
        if( head != (unsigned char)(tail-1) ) /* cast needed to avoid
promotion to integer */
            buf[ head++ ] = c;                /* access to a 256 byte aligned
array */
    }
```

В примере оптимизированной функции хорошо видно, что для прямого включения ассемблерного кода необходимо использовать директивы *_asm* и *_endasm*.

```
void to_buffer_asm( unsigned char c )
{
    _asm
    mov  r2,dpl
;buffer.c if( head != (unsigned char)(tail-1) ) /* cast needed to avoid
promotion to integer */
    mov  a,_tail
    dec  a
    mov  r3,a
    mov  a,_head
    cjne a,ar3,00106$
    ret
00106$:
;buffer.c buf[ head++ ] = c; /* access to a 256 byte aligned array */
    mov  r3,_head
    inc  _head
    mov  dpl,r3
    mov  dph,#(_buf >> 8)
    mov  a,r2
    movx @dptr,a
00103$:
    ret
    _endasm;
}
```

Внутри ассемблерной вставки возможно использование любых директив,

понятных ассемблеру.

Использование меток

Внутри функции можно определять метки вида nnnn$, где n – число от 0 до 100. Метки, используемые в языке Си, не видны внутри ассемблерных вставок и наоборот. Метки в ассемблерных вставках внутри разных функций также не видны друг для друга.

```
foo() {
        /* Некоторый код на Си */
        _asm
          ; Некоторый ассемблерный код
          ljmp 0003$
        _endasm;
        /* Еще код на Си */
    clabel:  /* Встроенный ассемблер не видит эту метку */ 3.6
        _asm
        0003$: ; Эта метка доступна только из встроенного ассемблера
        _endasm ;
        /* Еще код на Си */
    }
```

Директива __naked

Директива *__naked* позволяет исключить генерацию вводной части функции. Предполагается, что за сохранение контекста отвечает программист.

```
volatile data unsigned char counter;

    void simpleInterrupt(void) __interrupt (1)
    {
        counter++;
    }

    void nakedInterrupt(void) __interrupt (2) __naked
    {
        _asm
          inc     _counter ; Инкремент не меняет флагов,
                           ;  нет необходимости сохранять psw
          reti             ; Неоходимо явно указывать reti
        _endasm;
    }
```

Без *__naked* получается такой код:

```
_simpleInterrupt:
    push    acc
    push    b
    push    dpl
    push    dph
    push    psw
    mov     psw,#0x00
    inc     _counter
    pop     psw
    pop     dph
    pop     dpl
    pop     b
    pop     acc
    reti
```

С *__naked* код выглядит так:

```
_nakedInterrupt:
    inc    _counter ; Инкремент не меняет флагов,
                    ; нет необходимости сохранять psw
        reti        ; Неоходимо явно указывать reti
```

5.6. Объектные модули

Рассмотрим объектный модуль в контексте императивного программирования (модели вычислений Фон-Неймана).

Обьектный модуль (также – объектный файл, англ. object file) – это файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки исходного кода компилятором. Объектный файл содержит в себе особым образом подготовленный код (часто называемый бинарным), который может быть объединён с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполняемого модуля либо библиотеки.

Объектный модуль – согласно ГОСТ 19781-90 – программный модуль, получаемый в результате трансляции исходного модуля.

Объектный модуль – программа на машинном языке с неразрешенными внешними ссылками.

Компоновщик собирает код и данные каждого объектного модуля в итоговую программу, вычисляет и заполняет адреса перекрестных ссылок между модулями. Также в процессе компоновки происходит связывание программы со статическими и динамическими библиотеками (являющихся архивами объектных файлов).

Компиляторы и редакторы внешних связей создают выполняемые объектные файлы, предназначенные для запуска на определенных компьютерах. В случае использования кросс-компиляторов, на одном компьютере компилируются и редактируются объектные файлы, предназначенные для

выполнения на другом компьютере. Термин целевой компьютер обозначает тот компьютер, на котором предполагается выполнять объектный файл. За редким исключением целевой компьютер – это в точности тот же компьютер, на котором создается объектный файл.

Состав объектного модуля

1. Секции машинного кода с неопределенными адресами переходов.
2. Секции данных (в виде массивов констант и информации о размере областей ОЗУ).
3. Таблицы, содержащие имена функций, с указанием типов параметров.
4. Таблицы, содержащие имена глобальных переменных, с указанием типа.
5. Дополнительная отладочная информация. Как правило, это имена функций, локальных переменных, информация о местоположении параметров функций внутри фрейма стека, имена файлов с исходными текстами и номера строк.

Секции

Секция есть наименьшая часть объектного файла, которая подвергается перемещению и рассматривается как нечто отдельное и различимое. Как правило, в объектном файле присутствуют три секции, содержащие в себе исполняемый код (TEXT), данные (DATA) и неинициализированные данные (BSS). В зависимости от реализации, названия секций могут отличаться. В других секциях могут размещаться комментарии, дополнительные сегменты команд и данных, разделяемые сегменты данных. Допускаются секции, определенные пользователем. Количество, тип и порядок расположения секций определяется типом формата и реализацией линкера.

Загрузочный модуль в контексте вычислительной модели Фон-Неймана – файл, полученный после компиляции и сборки проекта в формате, требуемом для выполнения на конкретной платформе.

В качестве загрузочного модуля также можно рассматривать конфигурационный файл для FPGA или CPLD.

К загрузочным модулям можно отнести COFF, ELF и HEX.

Формат COFF

COFF (common object file format) – спецификация формата объектных и исполняемых файлов, а также разделяемых (динамических) библиотек. Используется в основном в Unix системах и их производных. На основе этого формата фирма Microsoft сделала свой формат PE (Portable Executable) используемый в операционных системах Windows NT и более поздних.

Особенности COFF.

- В объектные файлы можно добавлять свою информацию, не нарушая работоспособности стандартных средств обработки подобных файлов.
- Отведено место для отладочной информации.
- Можно влиять на способ создания объектных файлов, используя директивы времени компиляции.

5.7. Формат ELF

ELF – исполняемый и объектный формат (Executable and Linkable Format). В этом формате хранятся исполняемые и объектные файлы, генерируемые большинством компиляторов для *nix систем. Для работы с файлами в формате ELF вам могут понадобится утилиты readelf и objdump, обычно входящие в дистрибутивы linux или в состав пакета CYGWIN. С помощью этих программ вы можете посмотреть содержимое различных разделов объектных файлов.

Необходимо заметить, что для запуска программы, представленной в формате ELF нужен специальный загрузчик, понимающий этот формат. Такие загрузчики есть в достаточно больших системах, например таких, как Linux. Что же делать, если компилятор выдает формат ELF, а нам нужно запустить программу на «голом железе»? Первый вариант – реализовать в контроллере загрузчик ELF программ. Исходные тексты такого одного из вариантов такого загрузчика (к примеру, binfmt_elf.c) можно найти в eCos, Linux, FreeBSD и многих других системах. Если внимательно почитать документацию на формат, то ничего сложного в реализации этого загрузчика нет. Вам придется скопировать куски кода и данных из загрузочного модуля, переписать их в ОЗУ по определенным адресам, произвести инициализацию адресов и передать управление на начало программы. Загрузчик ELF очень похож на аналогичный загрузчик EXE программ от Microsoft.

Второй вариант – воспользоваться утилитой преобразования в бинарный или HEX файл, например, при работе с процессорами на базе ARM, утилитой arm-elf-objcopy, входящей в состав пакета GNUARM. Этот вариант значительно проще в реализации, но менее удобный и гибкий, так как предлагает жесткую привязку программы к указанному физическому адресу (ELF можно было бы запускать где угодно в памяти).

Формат Intel HEX

Intel HEX – формат файла. Основным отличием этого формата от таких монстров, как ELF и COFF является крайняя простота. Формат позволяет хранить только образ памяти. Ни о каком перемещаемом коде и возможности хранения объектных файлов в этом формате речи не идет.

В настоящий момент этот формат в основном используется при

программировании встроенных систем. Большинство компиляторов и линкеров умеют выдавать загрузочный модуль в этом формате. Строки файла представляют собой текстовые записи, в которых закодированы адреса расположения, команды и данные в шестнадцатеричной системе счисления. Изначально, HEX формат использовался для работы с перфоленточными загрузчиками. В настоящее время HEX формат используют для программирования различных контроллеров и связи с программаторами ППЗУ.

Каждую строку в HEX файле называют записью. Она состоит из следующих элементов:

1. Двоеточие (:).
2. Число байтов данных, содержащихся в этой записи. Занимает один байт (две шестнадцатеричных цифры), что соответствует 0…255 в десятичной системе.
3. Начальный адрес блока записываемых данных – 2 байта. Этот адрес определяет абсолютное местоположение блока в EPROM.
4. Один байт, обозначающий тип записи.
    1. 0x00 – блок данных.
    2. 0x01 – конец файла.
    3. 0x02 – адрес сегмента (см. архитектуру процессора Intel x86).
    4. 0x03 – стартовый адрес сегмента (см. архитектуру процессора Intel x86).
    5. 0x04 – старшая часть линейного (32 - разрядного) адреса
    6. 0x05 – стартовый адрес, старшая часть линейного (32 - разрядного) адреса.
5. Байты данных (их число указывается в поле 2).
6. Последний байт в записи является контрольной суммой. Если сумма всех байтов в строке (без учёта переноса) равняется 00, строка считана правильно.
7. Строка заканчивается стандартной парой CR/LF (0Dh 0Ah).

Файл всегда завершается командой 01, (получается запись вида «:00000001FF»).

Пример программы для вычисления контрольной суммы

```
//-------------------------------------------------------------
// Проверка контрольной суммы:
// ВХОД: адрес HEX блока (с ':')
// ВЫХОД: 0 если норма, 1 если ОШИБКА
//-------------------------------------------------------------

int check_cs( unsigned char * hex_block )
{
unsigned char cs = 0;
int i;
int len;
```

```
    len = char_hex_bin( ++hex_block );

    for( i = 0; i < len + 4; i++ )
    {
        cs += char_hex_bin( hex_block );

        hex_block++;
        hex_block++;
    }
    cs = ~cs + 1;

    if( cs != char_hex_bin( hex_block ) )
         return ERROR;
    else
         return OK;
}
```

Пример HEX файла

```
:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

Недостатки формата

- Ненадежный контрольный код (вероятность ошибки 1/256).
- Нет суммарного контрольного кода для всего файла.
- Получается большой файл при HEX кодировании, что отрицательно сказывается на скорости передачи файлов в контроллер.

5.8. Компоновщик

Компоновщик, редактор связей, линковщик, линкер – средство объединения нескольких входных объектных модулей в единый выходной модуль.

В контексте программирования, в рамках модели вычислений Фон-Неймана, компоновщик является программой для создания исполняемой программы (загрузочного модуля) из нескольких объектных файлов.

Редактор связей ld является компоновщиком объектных файлов, входящим в состав GNU Binutils. Используется совместно с GCC.

GNU Binutils – набор утилит для сборки, ассемблирования, просмотра файлов и так далее.

- ld – редактор связей (компоновщик, линкер).
- as – ассемблер.

- addr2line – Выдает имя файла и номер строки исходного текста по адресу.
- ar – утилита для создания, изменения и извлечения из архивов.
- c++filt – декодирование низкоуровневых имен C++ (DeMangle).
- dlltool – создание файлов для создания и использования DLL.
- gprof – выводится информация профилировщика.
- nlmconv – конвертирует объектный файл в формат NLM (Netware Loadable Module).
- nm – вывод списки символов из объектных файлов.
- objcopy – копирует содержимое одних объектных файлов в другие.
- objdump – показывает информацию об объектном файле.
- ranlib – создает индекс к содержимому архива и сохраняет его в архиве.
- readelf – распечатывает информацию из объектного ELF файла.
- size – показывает размер разделов и общий размер для каждого объектного файла.
- strings – печатает список печатных (распечатываемых) строк из файла.
- strip – удаляет символы.
- windmc – компилятор сообщений (messages) совместимый с Windows.
- windres – компилятор файлов ресурсов Windows.

5.9. Библиотеки языка Си для встроенных систем

Библиотека языка Си (libc) является стандартизированным набором функций, реализующих простейшие механизмы для работы с памятью, операции ввода-вывода, математические функции и строковые операции. В чем особенность такой библиотеки с точки зрения встроенных систем? Во-первых, библиотека языка Си для встроенной системы должна быть компактной. Во-вторых, из-за того, что во встроенных системах применяются не самые быстрые процессоры, библиотечные функции должны выполняться быстро. Немаловажным фактом также является реентерабельность функций, позволяющая использовать функции в обработчиках прерываний и в многозадачной среде ОС РВ. К примеру, большинство доступных библиотек в старой операционной системе MS-DOS (которая еще пока применяется во встроенных системах) не отличается этим свойством. Аналогичные проблемы есть и в других библиотеках, например, в MicroLib, о которой пойдет речь ниже.

Стандартные библиотеки языка Си обычно являются спутниками сравнительно крупных операционных систем реального времени (к примеру таких, как eCos) или поставляются совместно с компилятором языка Си (например, библиотеки есть в SDCC или GNUARM).

В системах общего назначения, на первое место ставится переносимость и удобство, а не эффективность реализации. Поэтому, обычно, простой перенос

библиотек общего назначения приводит к тому, что библиотечные функции работают недостаточно быстро и съедают весьма существенный объем FLASH памяти контроллера.

На сайте фирмы Keil Software помещено сравнение библиотеки MicroLib, входящей в состав коммерческого пакета RealView, предназначенного для разработки ПО для процессоров с ядром ARM, со стандартной библиотекой языка Си для GCC.

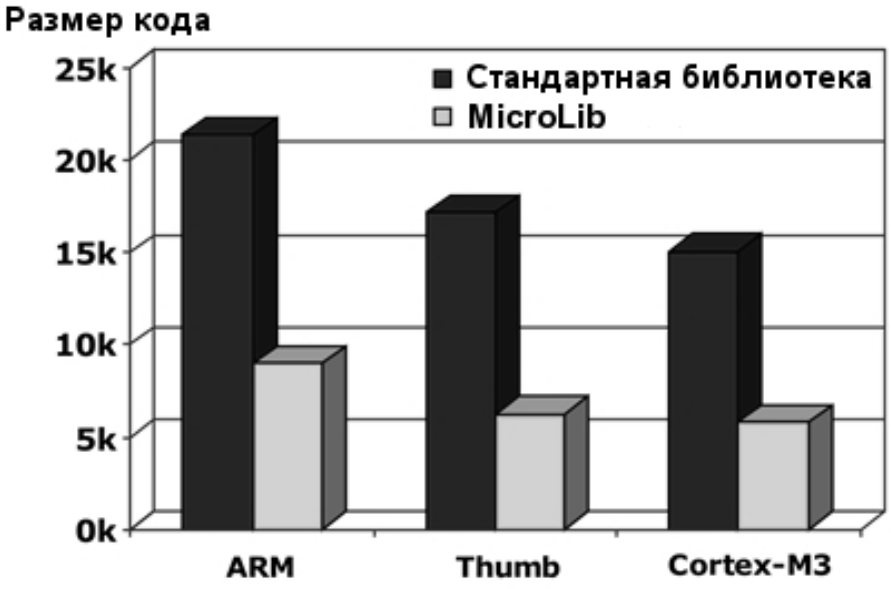


Рисунок 11 Сравнение размера кода для MicroLib и стандартной libc

К сожалению, за компактность нужно платить. Приведем перечень отличий MicroLIb от стандартной библиотеки Си (информация о библиотеке взята в конце 2008 года).

- Нет полной совместимости со стандартами ISO и IEEE 754.
- Не поддерживается locale (locale отвечает за интернацианализацию, формат времени и т.п.).
- Не работают функции abort(), exit(), atexit(), clock(), time(), system() и getenv().
- Не поддерживаются двухбайтовые строки (unicode).

- Код библиотеки нереннтерабельный, его использование в ОС РВ весьма проблематично.
- Не поддерживаются сигналы.

Более подробную информацию о MicroLib можно получить на сайте http://www.keil.com.

Использование библиотеки

В целом, библиотека языка Си достаточно слабо связана с аппаратным обеспечением, на котором вы хотите запускать свою программу. Этот факт нам очень облегчает жизнь при освоении очередного контроллера, сделанного на новом для нас процессорном ядре с какой-нибудь экзотической периферией. Исключение составляют функции ввода-вывода. Так как стандартная библиотека Си выросла из среды Unix, в ней весь ввод-вывод основан на концепции файла, над которым можно производить ряд несложных операций:

- открыть (open);
- создать (create);
- прочитать (read);
- записать (write);
- переместить указатель текущей позиции (lseek);
- закрыть (close).

В виде файлов, в Unix представлены все устройства ввода-вывода. Некоторые устройства ввода-вывода, например, такие как жесткие диски, позволяют осуществлять произвольный доступ (то есть менять позицию файле). Некоторые устройства, такие как терминал или последовательный канал позволяют производить только последовательный доступ к файлу.

Вспомним учебник по языку Си. Обычно первой программой является примерно следующий код:

```
#include <stdio.h>

int main( void )
{
    printf(«Hello, world!\n»);
    return 0;
}
```

Как нам известно из документации, библиотечная функция **printf** осуществляет вывод строки в поток (stream) стандартного вывода (stdout). Стандартный вывод, а также вывод ошибок (stderr) попадают на то, что мы обычно называем экраном. Точнее, это можно было бы назвать консолью или терминалом. Клавиатурный ввод обычно попадает в поток стандартного ввода (stdin). Как нам реализовать работу с функциями **printf**, **putchar** и **getchar** в контроллере, у которого и экрана как такового нет, и клавиатуры тоже? Ответ

зависит от реализации библиотеки.

В некоторых библиотеках, например, в библиотеке, поставляемой в составе пакета softune фирмы Fujitsu, для реализации ввода-вывода необходимо самостоятельно реализовать функции open, close, read, write, sbrk. В некоторых, например, таких как SDCC, достаточно реализовать функции putchar и getchar.

```
#include <stdio.h>


void putchar( char c )  { wsio( c );     }
char getchar( void )    { return rsio(); }
```

Первый вариант более гибкий, так как позволяет реализовать не только терминальный ввод-вывод, но и обращение к внешней памяти с произвольным доступом. Второй вариант значительно проще в реализации и очень хорошо подходит к маломощным контроллерам, для которых собственно и сделан SDCC.

Для того, чтобы правильно сделать привязку системы ввода-вывода к вашей аппаратуре необходимо лучше понимать суть отличий между функциями open, write, read, close и их аналогами fopen, fwrite, fread, fclose. Отличие состоит не только в типе дескриптора и разных способов работы с двоичными и текстовыми файлами. Отличия несколько глубже. Основная суть отличия файловых операций состоит в том, что в случае с fwrite/fread происходит дополнительная буферизация, а при работе функций write/read теоретически никакой буферизации нет (на самом деле, в большинстве операционных систем она есть, да и вам её никто не мешает сделать). Когда вы выводите строку «Hello, World!» на экран с помощью функции printf, строка не появится, если вы не добавите в конце символа «/n». Исключение составляют опять таки некоторые реализации библиотек, к примеру, для MS-DOS. Сброс буфера (то есть фактический вызов низкоуровневой функции write) происходит обычно по трём событиям:

- по факту вызова функции **fflush**;
- по заполнению буфера;
- при появлении символа /n (в текстовом режиме).

Хочется сразу заметить, что при «сыром» (raw) вводе-выводе никаких текстовых или двоичных режимов нет.

5.10. Утилита make

Утилита make, как известно, используется для сборки сравнительно больших проектов. Раньше, когда компьютеры не обладали такими вычислительными мощностями как сейчас, время компиляции исходных текстов было настолько велико, что возникла необходимость в раздельной и выборочной компиляции. В чём суть этой раздельной компиляции? Процесс

разбивался на два этапа: превращение исходных текстов в объектные модули и сборка этих объектных модулей в загрузочный модуль. Далее, если производить компиляцию выборочно, то есть если не компилировать все исходные тексты подряд, а компилировать только те, которые претерпели изменения, можно получить существенную экономию времени сборки всего проекта.

Для того, чтобы прочувствовать необходимость такой выборочной компиляции сейчас, пожалуй необходимо скомпилировать что-то действительно большое. К примеру, ядро операционной системы Linux, или собрать из исходных текстов Open Office. Правда, помня о такой маркетинговой уловке производителей железа как закон Мура, я полагаю, с неуклонным ростом вычислительных ресурсов этот пример станет не очень удачным уже достаточно скоро.

Зачем же нам тогда make, спросите вы? Ведь большинство проектов для встроенных систем весьма аскетичны, а скомпилировать три десятка модулей на C или C++ современный компьютер может за считанные секунды?

Ответ очень прост: make достаточно мощный, *специализированный* инструмент для работы с проектами.

Со специализацией инструментов я столкнулся в полной мере, когда попробовал использовать язык Perl для обработки текстов вместо обычного на тот момент для меня C++. Результат меня тогда поразил. За три дня на язке Perl был выполнен объем работы, которые потребовал от меня две недели на C++! Это при том, что C++ я уже использовал давно, а Perl видел в первый раз в жизни. Perl специально создавали как средство для обработки текстов. C++ не помогла библиотека STL, хотя вроде как набор механизмов у инструментов был примерно одинаковый. Разве что в C++ у меня не использовались библиотеки для разбора регулярных выражений.

В чём же оказалась сила Perl? В первую очередь – в простоте. Краткая синтакическая конструкция, вызывающая у новичка головную боль и мысли о распечатке дампа памяти, делает то, что на C++ потребует написания полутора десятков строк текста. Нам известно, что уровень языка не влияет на количество ошибок. Влияние оказывает количество строк, то есть чем больше строк, тем больше ошибок и наоборот. В программе на Perl строк было мало и я продвигался к завершению проекта быстрыми темпами.

Итак, вернемся к make. Основная цель утилиты – сократить время, необходимое для описания того, что надо сделать с проектом. Предполагается при этом, что вы работаете с проектом самостоятельно, без каких либо IDE.

Что нужно делать с проектом во время сборки?

- Убрать старые и ненужные файлы, объектные модули, всякий мусор, исполняемые файлы и т.д. Это полезно делать перед архивацией проекта или перед отдачей проекта системе контроля версий.
- Запаковать проект в архив, пр этом желательно переписать его в пару-тройку мест, а чтобы не запутаться – назвать архив так-же, как называется проект.

- Скомпилировать исходные тексты.
- Собрать загрузочный модуль.
- Вызвать необходимые конвертеры для получения формата, удобоваримого для доставки в контроллер.
- Вызвать программу для доставки загрузочного модуля в контроллер.

Ко всему этому хочется добавить, что проектов у вас много, а вы один. Большинство описанных выше действий вполне себе стандартно и хочется вынести их за скобки. Утилита make позволяет описать параметры сборки проекта в виде переменных, а из этих переменных задать последовательность действий. Получается, что переменные от проекта к проекту практически не меняются, а вот их содержимое может изменяться.

Что мы можем указать в виде таких переменных? Попробуем привести список.

- Имя проекта. Это имя можно использовать как имя архива, а также как имя исполняемого или загрузочного модуля.
- Дополнительные части для имени проекта, такие как дата и время компиляции, порядковый номер сборки, тип проекта.
- Имя компилятора. В области встроенных систем считается вполне нормальным частый переход на разные микроконтроллеры и, соответственно, использование разных компиляторов раз в несколько месяцев.
- Опции компилятора. Флаги оптимизации, специфические опции, местоположение заголовочных файлов и т.п.
- Опции линкера. Наименование библиотек, адреса памяти программ и кода и т.д.
- Архиватор и ключи для архивации.
- Имя программы и ее ключи для доставки ПО в контроллер.

Список этот далеко не полный, его можно продолжать.

Использование make

По своей структуре makefile (обычно так называют программу для make) можно разделить на две части: описание переменных и описание действий над этими переменными. Программа для make очень напоминает форму Бэкуса-Наура (БНФ, по английски Backus-Naur form, BNF), позволяющую описать синтаксис языка программирования. В БНФ одна конструкция последовательно определяется через другие. Такую запись, как в make, можно увидеть в знаменитом компиляторе компиляторов YACC. Для того, чтобы вам было проще понять make, я вам рекомендую почитать литературу про БНФ.

С переменными все просто, формат записи следующий:

```
PROJECT = test_sdk11
```

Для использования переменной достаточно указать имя переменной в скобках со знаком доллара:

```
$(PROJECT)
```

Вместо PROJECT будет подставлено test_sdk11.

Часть makefile, отвечающая за действия, состоит из ряда правил, каждое из которых имеет следующий вид:

```
Цели: зависимость1 зависимость2 ...
    команда 1
    команда 2
    ...
```

Цель (target) – это то, ради чего существует данное правило. Если это файл, то утилита make учитывает факт его наличия и время его создания. Зависимости показывают, от чего данное правило зависит. Любое изменение зависимости приведет к запуску команды. Зависимости могут быть описаны ниже как цели. Команды – перечень команд, которые исполняются при исполнении правила. В общем случае их может и не быть, так же как может и не быть зависимостей.

Пример правила без зависимостей:

```
clean:
        -rm -f  $(NAME).hex \
                $(NAME).bin \
                $(NAME).map \
                $(NAME).lst
```

В данном правиле производится стирание файлов, имя которых состоит из содержимого переменной NAME и расширений .hex, .bin, .map и .lst. Необходимо заметить, что оно будет работать совершенно неожиданным образом, если в каталоге будет находится файл с именем clean.

Значение переменных в makefile задается с помощью знака присваивания. В переменную попадает все содержимое строки.

```
LFLAGS  = --code-loc 0x2100 --xram-loc 0x6000 --stack-loc 0x80
```

Возможно продление строки с помощью обратного слэша. Кроме того, возможно использование ранее определенных переменных при определении новых.

```
LIST_SRC = \
$(SRC_DIR)/led.c \
$(SRC_DIR)/max.c \
$(SRC_DIR)/test_led.c
```

В круглых скобках возможно производить различные действия, с помощью достаточно большого количества функций. Например, возможна переделка списка исходных текстов в список объектных модулей:

```
LIST_OBJ = $(LIST_SRC:.c=.rel)
```

С помощью функции shell возможен вызов командного интерпретатора. Результатом его работы является строка, полученная через стандартный вывод и присваиваемая переменной.

```
DATE     = $(shell date +%d-%m-%g_%H-%M-%S)
VERSION  = $(shell cat VERSION)
```

В документации к утилите make подробно описаны функции, которые можно использовать при определении переменных.

5.11. Система контроля версий

Система контроля версий решает сразу несколько задач для разработчика встроенного ПО.

- Позволяет сохранять в надёжном месте весь его проект (выполняет функции интеллектуального архиватора).
- Позволяет предоставить доступ к проекту для всех его участников, а также регламентировать доступ для посторонних людей.
- Позволяет отслеживать все изменения в проекте, возвращаться к старым версиям файлов и вести параллельно сразу несколько вариантов одного проекта.

Большинство опытных монстров и зубров в области программирования могут возразить, что для сравнительно несложных программных проектов, существующих в области встроенных систем, система контроля версий – лишняя бюрократическая процедура, отнимающая время и не дающая ничего взамен. Хочу вас заверить, что это не так. Даже если вы работаете один, и вам не нужно совместно с кем-то работать над одним проектом, система контроля версий оказывается гораздо удобнее банального архиватора. С помощью такой, к примеру, команды ***svn co http://vash_server.vash_domen.ru/repos/proect*** вы можете получить доступ к своим исходным текстам везде, где есть компьютеры и выход в Интернет. С помощью простой команды ***svn commit*** вы можете отправить свои изменения обратно на сервер. Вам не нужно больше бояться,

что вирусы уничтожат ваши файлы, или что у вас сломается жесткий диск. Все данные хранятся на удаленном сервере и всё, что вы можете потерять, – это несколько часов работы. Естественно, все это будет работать именно так, если вы позаботитесь о регулярном резервном копировании вашего сервера.

В настоящее время системы контроля версий интегрированы в различные интегрированные среды разработки (IDE). Наиболее распространенными системами контроля версий, широко применяемыми на момент написания этой книги, являются CVS и Subversion. Эти системы очень похожи. CVS появилась раньше, чем Subversion и хорошо себя зарекомендовала. Subversion в настоящий момент активно пробивается в лидеры, в ней исправлено несколько концептуальных ошибок, имеющихся в ее предшественнике SVN. Чем пользоваться вам – решайте сами. По сути, все системы контроля версий очень похожи и чем пользоваться – дело привычки или каких-то иных предпочтений.

Работа с системой контроля версий

Работа с системой контроля версий начинается с создания репозитория. Репозиторий – это специализированная база данных, в которой хранятся файлы вашего проекта, вносимые изменения, информация о создании и удалении файлов, информация о пользователях и т.п. В основном, репозиторий рассчитан на хранение обычной текстовой информации, хотя можно хранить и двоичные файлы (к примеру, исполняемые, документацию в формате PDF или DOC). Почему не использовать обычную СУБД, к примеру, такую как MySQL или Oracle? Ответ прост: проблема в эффективности. Репозиторий системы контроля версий специально ориентирован на хранение текстовых документов и изменений в них. Хранение таких данных (весьма большого объема) в обычной базе данных вызовет неэффективное использование дискового пространства и замедление работы. Обычно в репозитории хранится разница между текущей и предыдущей версией фала. Для того, чтобы посмотреть на то, как может выглядеть внутренности репозитория, изучите работы программы ***diff***.

После создания репозитория можно добавлять и стирать в нем файлы, получать содержимое репозитория на свою машину, проверять обновления и отправлять изменения обратно. Типичный цикл работы на примере Subversion может выглядеть примерно так:

```
svn co http://194.85.162.173/repos/ul3
```

Эта строка позволяет забрать с сервера 194.85.162.173 из репозитория repos/ul3 ваш проект. Вы можете отредактировать нужные вам файлы. Для того, чтобы убедиться, что ваши товарищи не добавили в проект чего-то нового можно вызвать команду:

```
svn update
```

Исполнение этой команды приводит к тому, что вы забираете из репозитория все новые файлы, появившиеся там после того, как вы скачали версию себе на машину. Если вы умудрились редактировать с кем-то один и тот-же файл, система контроля версий должна предупредить вас о конфликте. Разрешение конфликта делается в соответствии с документацией на используемый вами инструмент.

Для записи изменений, сделанных вами в репозиторий воспользуйтесь следующей командой:

```
svn commit
```

Данная команда отправит в репозиторий только те файлы, которые вы изменили. Если файлы были изменены кем-то еще вам будет сообщено о конфликте. В процессе запуска команды будет вызван текстовый редактор и вам будет предложено описать сделанное изменение. Пожалуйста не ленитесь писать по существу. Эти записи потом вам сильно помогут разобраться в проекте, когда пройдет существенное время и вы все основательно подзабудете.

О более сложных вариантах использования систем контроля версий я рекомендую читать самостоятельно в специальной литературе.

6. Отладка ПО ВВС

6.1. Основные определения

Отладка – доведение до работоспособного состояния системы в процессе ее проектирования.

Отладка – деятельность, направленная на установление точной природы известной ошибки, а затем, на исправление этой ошибки.

Отладчик (англ. Debugger) – инструментальное средство, помогающее найти ошибку.

6.2. Специфика отладки ПО встраиваемых систем

Основной особенностью встраиваемых систем является тот факт, что отлаживаемая программа запускается не в среде разработки, а на удаленной машине. Кроме того, встраиваемое ПО тесно связано с окружающим аппаратным обеспечением.

Существует два способа организации отладки ПО встраиваемых систем:

1. погружение отлаживаемого ПО в симуляционную среду;

2. внедрение отладочного агента в целевую систему.

Оба подхода имеют свои достоинства и недостатки. Создание симулятора, очень точно имитирующего реальные устройства, весьма проблематично, поэтому модель и реальное оборудование будут всегда отличаться. Кроме того, номенклатура выпускаемых микроконтроллеров и различных периферийных устройств (таких как микросхемы памяти, ЦАП, АЦП, сетевые контроллеры, ЖКИ и т.п.) так велика, что производители симуляторов просто физически не способны воспроизвести такое обилие библиотечных элементов.

У внедрения отладочных агентов тоже есть свои проблемы. Для начала, рассмотрим варианты организации системы отладки второго типа.

1. Программный отладочный агент, взаимодействие с отладчиком через какой-либо интерфейс (к примеру, RS-232 или Ethernet).
2. В качестве отладочного агента выступает специальная эмулирующая головка, позволяющая заменить микроконтроллер целиком. Обычно, такие системы называют внутрисхемными эмуляторами.
3. Отладочный агент встроен аппаратно в микроконтроллер. В настоящее время самым распространенным вариантом такого агента является JTAG.

Все три варианта дают примерно одинаковый результат:

- возможность отладки в исходных текстах;
- просмотр регистров, стеков и памяти;
- работа с точками остановки;
- пошаговая отладка.

Программный отладочный агент самый простой и дешевый. При этом он самый медленный и ненадежный. Более хорошие результаты получаются с использованием внутрисхемными эмуляторами и JTAG. Внутрисхемный эмулятор является наиболее дорогим, но зато и наиболее мощным средством отладки.

Интерфейс JTAG значительно дешевле и универсальнее, так как нет необходимости покупать эмулятор для каждого микроконтроллера. К сожалению, JTAG значительно проигрывает внутрисхемным эмуляторам в скорости работы.

6.3. Инструментальные средства отладки

Симулятор

Симулятор – система для полной или частичной имитации поведения и структуры какого-либо объекта. Симулятор относится к инструментальным

средствам отладки, тестирования и верификации программных и аппаратных компонент вычислительной системы.

В случае программной реализации симулятор можно исполнять на инструментальной машине. В программировании встроенных систем и СнК чаще всего используется симулятор процессора. Внешне такой симулятор выглядит как обычный отладчик.

При симуляции различных ASIC, процессоров, графических акселераторов используют программно-аппаратные симуляторы, выполненные на базе FPGA. Такой подход позволяет существенно сократить время тестирования и верификации по сравнению с чисто программной симуляцией. Стоимость программно-аппаратного симулятора значительно выше стоимости программного.

К достоинствам симулятора можно отнести возможность моделирования окружающей среды исследуемой системы, что позволяет производить работу в реальном масштабе времени с точки зрения наблюдателя, находящегося внутри исследуемой системы.

К сожалению, в симуляторах всегда присутствует инструментальная погрешность, возникающая из-за неточности моделирования, абстракции, ошибок реализации и так далее.

Внутрисхемный эмулятор

Внутрисхемный эмулятор – это аппаратное устройство, используемое при отладке, обычно выполняемое в форме микропроцессора с дополнительными контактами.

Внутрисхемные эмуляторы подключаются к отлаживаемой или тестируемой системе вместо целевого микропроцессора или микроконтроллера и позволяют гибко управлять поведением системы на протяжении процесса отладки, собирать данные о состоянии ее различных объектов, выполнять программы пользователя в различных режимах: в режиме реального времени (непрерывное выполнение программы с заданного адреса), в пошаговом режиме, в режиме с остановками функционирования по заданному условию. Зачастую они позволяют эмулировать не только целевой процессор, но и память, тактовый генератор, устройства ввода-вывода.

Применение внутрисхемных эмуляторов позволяет решить почти все проблемы, связанные с отладкой и тестированием программного обеспечения и аппаратуры. К сожалению, крупным недостатком внутрисхемных эмуляторов является их очень высокая стоимость.

Внутрисхемные эмуляторы являются средством, заменяющим на аппаратном уровне часть целевой системы. В настоящее время распространены

два основных варианта:

- эмулятор процессора;
- эмулятор ПЗУ.

В последнее время, с появлением ОКМЭВМ со встроенной постоянной памятью (в виде FLASH или OTP), второй вариант эмуляторов начал постепенно выходить из употребления. В принципе, внутрисхемный эмулятор имеет тот же набор функций, что и программный симулятор. Приведем основные отличия:

- отладка возможна на реальном оборудовании (что не исключает возможности программной имитации окружения);
- отладка может производиться в реальном времени.

Интересным вариантом внутрисхемной эмуляции является JTAG. В классических эмуляторах процессора для проведения отладки центральный процессор заменяется на эмулирующую головку (у ряда новых процессоров эмулирующую головку можно подключать непосредственно к впаянному кристаллу). Это приводит к необходимости ставить на плату панель под процессор, что уменьшает надежность системы. При использовании технологии JTAG, эмулятор подключается к плате через специальный технологический разъем. При этом процессор не вынимается. JTAG позволяет отключить ядро процессора и управлять шиной адреса, данных и управления напрямую. К сожалению, для управления всеми выводами процессора необходимо передавать через порт JTAG большое количество информации. Поэтому отладка в реальном времени (на частоте работы процессора) невозможна. Большим достоинством JTAG является аппаратная простота эмулятора. Например, в самом простом случае, достаточно подключить порт JTAG к параллельному LPT порту обычного ПК.

IEEE 1149.1 JTAG - механизм граничного сканирования

JTAG предназначен для решения следующего перечня основных задач:

- начальное тестирование, которое выявляет технологические дефекты изготовления;
- доставка необходимой конфигурации для программируемых компонент;
- поддержка различного рода отладочных механизмов (статических или динамических) и режима мониторинга.

Гибкость использования механизма граничного сканирования достигается рядом особенностей стандарта, основными из которых являются:

- возможность параллельной согласованной работы нескольких устройств, поддерживающих данный стандарт (количество ограничивается электрическими параметрами интерфейса);
- возможность расширения самого механизма (введение дополнительных

команд и форматов данных).

Первое определяет возможность использования механизма граничного сканирования при разработке многопроцессорных систем с гомогенной или гетерогенной структурой. Второе расширяет круг задач, где может эффективно применяться механизм, позволяя решать, например, задачи отладки, сбора диагностики и мониторинга.

Реализация JTAG-инструментария

Решая задачу схемотехнического проектирования встроенной системы, проектировщики закладывают средства начального тестирования и внутрисхемной инициализации. Для этого разрабатывается специализированное устройство или сервисный механизм инструментально-технологического характера. Функциональность и состав инструментального обеспечения определяются особенностями проекта. Инструментальные кросс-средства представляют собой совокупность программных средств разработки и аппаратных интерфейсов, которые обеспечивают доступ к целевому объекту с инструментальной машины. Ресурсов инструментальной машины обычно хватает для реализации интерпретатора языкового описания механизма граничного сканирования при проведении большинства работ с макетным образцом системы на этапах частичной инициализации и начальной отладки. Иначе обстоит дело с опытными образцами системы, когда при проектировании стараются сокращать выделенные инструментальные каналы, формируя объединенные сервисные механизмы. При этом начальная инициализация происходит с помощью ранее апробированных вычислительных компонент, которые совмещают в себе целевую функциональность и поддержку инструментального режима. Например, инструментальный канал и канал для поддержки механизма граничного сканирования могут быть объединены. В этом случае размещение интерпретатора на инструментальной машине приведет к снижению эффективности работы из-за ограничений канала. Актуальной становится реализация интерпретатора средствами “сервисного” контроллера в составе целевой системы, который будет выполнять резидентную программу поддержки механизма граничного сканирования. Решение в пользу такой резидентной реализации интерпретатора будет зависеть от вычислительной мощности контроллера и сложности интерпретируемого языка. На сегодняшний день многие производители предоставляют такого рода интерпретаторы в исходных текстах, для различных аппаратных платформ. Типовой вариант JTAG – системы приведен на рис. 5. Эффективность использования технологии граничного сканирования во многом зависит от решения двух вопросов: описания структуры цепочки JTAG и формулировки алгоритма работы с описанной цепочкой. Под JTAG-цепочкой (scan path) понимается полная стандартная тест-шина, полученная последовательным соединением сигналов TDI и TDO нескольких компонент. Это понятие

используется при решении следующих задач:

- описания отдельных микросхем, поддерживающих механизм граничного сканирования;
- описания структуры целевой системы с точки зрения механизма граничного сканирования;
- описания (выделения) цели работы конкретного алгоритма (программирование, тестирование).

Обычно в реальной системе имеется несколько микросхем, связанных в одну JTAG-цепочку. Как отмечалось выше, для описания цепочек и их иерархии используются языки BSDL и HSDL. Как средства описания они эффективны, но практически не поддержаны доступными инструментальными средствами. В результате цепочку JTAG-устройств разработчик задает тем или иным неформальным образом, что ведет к резкому росту трудоемкости при использовании механизма граничного сканирования. Задача выделения цели не входит в число стандартных задач описания структуры JTAG-цепочки. Однако на практике приходится очень редко работать с регистром граничного сканирования BSR (Boundary Scan Register), равным объединению BSR всей цепочки. Обычно для конкретного теста или алгоритма требуется не более десятка ячеек в BSR или только одна микросхема в составе цепочки. Стандартного средства такого описания, по-видимому, не существует. Это объясняется, скорее всего, спецификой описания (по факту) JTAG-цепочки и кругом решаемых JTAG задач. Есть упоминания о средствах программирования, например, flash-памяти, с помощью JTAG, что требует выделения линий адреса, данных и управления во всем BSR. Однако в известных реализациях это делается неформальным образом. Справедливости ради, необходимо отметить, что в стандарте HSDL имеется возможность «выделять» отдельные ячейки BSR, назначая им произвольные имена. Но это нельзя назвать выделением цели в чистом виде, поскольку алгоритм, составленный по такому описанию, по-прежнему будет оперировать полным BSR. Что касается разработчика, то ему при работе также придется рассматривать всю JTAG-цепочку (явным или неявным образом), даже если используется только несколько выводов одной из микросхем.

Измерение производительности программ

Профилировка (профилирование) – измерение производительности как всей программы в целом, так и отдельных ее фрагментов, с целью нахождения «горячих» точек (Hot Spots) – тех участков программы, на выполнение которых расходуется наибольшее количество времени.

Профилировщик (профайлер, profiler) – основной инструмент оптимизатора программ. Программный код ведет себя как в известной

пословице самый медленный верблюд, который определяет скорость каравана, то есть производительность приложения определяется самым узким его участком. Программисты нуждаются в инструментальных средствах, чтобы проанализировать их программы и идентифицировать критические участки программы.

Профилировщики помогают определить, как долго выполняются определенные части программы, как часто они выполняются, или генерировать дерево вызовов (call graph). Типично эта информация используется, чтобы идентифицировать те части программы, которые работают больше всего. Эти трудоёмкие части могут быть оптимизированы, чтобы выполняться быстрее. Это общая методика для отладки.

Цели и задачи профилировки

Основная цель профилировки – исследовать характер поведения программы во всех её точках. Под «точкой», в зависимости от степени детализации, может подразумеваться как отдельная машинная команда, так целая конструкция языка высокого уровня (например, функция, цикл или одна–единственная строка исходного текста).

Большинство современных профилировщиков поддерживают следующий набор базовых операций:

- определение общего времени исполнения каждой точки программы (total [spots] timing);
- определение удельного времени исполнения каждой точки программы ([spots] timing);
- определение причины и/или источника конфликтов и пенальти (penalty information);
- определение количества вызовов той или иной точки программы ([spots] count);
- определение степени покрытия программы ([spots] covering).

Общее время исполнения

Сведения о времени, которое тратится на выполнение каждой точки программы, позволяют выявить её наиболее «горячие» участки. Правда, здесь необходимо сделать одно уточнение. Непосредственный замер покажет, что, по крайней мере, 99,99% всего времени выполнения профилируемая программа проводит внутри функции main, при этом «горячей» является отнюдь не сама main, а вызываемые ею функции. Чтобы не вызывать у программистов недоумения, профилировщики обычно вычитают время, потраченное на выполнение дочерних функций, из общего времени выполнения каждой функции программы.

Удельное время выполнения

Если время выполнения некоторой точки программы не постоянно, а варьируется в тех или иных пределах (например, в зависимости от рода обрабатываемых данных), то трактовка результатов профилировки становится неоднозначной, а сам результат – ненадежным. Для более достоверного анализа требуется: а) определить действительно ли в программе присутствуют подобные «плавающие» точки и, если да, то: б) определить время их исполнения в лучшем, худшем и среднем случаях.

Определение количества вызовов

Оценивать температуру точки можно не только по времени ее выполнения, но и частоте вызова. Например, пусть у нас есть две «горячие» точки, в которых процессор проводит одинаковое время, но первая из них вызывается сто раз, а вторая – сто тысяч раз. Нетрудно догадаться, что, оптимизировав последнюю хотя бы на 1%, мы получим колоссальный выигрыш в производительности, в то время как, сократив время выполнения первой из них вдвое, мы ускорим нашу программу всего лишь на четверть.

Таким образом, часто вызываемые функции в большинстве случаев имеет смысл «инлайнить» (от английского in-line), т. е. непосредственно вставить их код в тело вызываемых функций, что сэкономит какое-то количество времени.

Определение степени покрытия

Покрытие – это процент реально выполненного кода программы в процессе его профилировки. Такая информация, в первую очередь, нужна тестерам, чтобы убедиться, что весь код программы протестирован целиком и в ней не осталось никаких «темных» мест. С другой стороны, оптимизируя программу, очень важно знать, какие именно ее части были профилированы, а какие нет. В противном случае, многих «горячих» точек можно просто не заметить только потому, что соответствующие им ветки программы вообще ни разу не получили управления.

Анализ исходного кода

Анализатор кода (code analyzer, code reviewing software) – программное обеспечение (ПО), которое помогает обнаружить ошибки (уязвимые места) в исходном коде программы.

Анализ кода (code analysis) – это близкий родственник обзора кода (code review).

Code review – это систематическая проверка исходного кода, проводящаяся для того, чтобы обнаружить ошибки, допущенные на стадии разработки,

улучшить качество программы и навыки разработчиков.

Обычно code review включает участие:

- человека, который код писал;
- человека (или людей), которые этот код могут читать и понимать, насколько хорошо он удовлетворяет общим и конкретным критериям.

Общие критерии представляют собой стандарты кодирования (coding standard). Конкретные критерии подразумевают знания требований, для удовлетворения которых код написан.

Процедура анализа кода отличается от тестирования. При тестировании программа проверяется на некотором наборе входных данных с целью выявления несоответствия действительного поведения программы специфицированному. Однако спецификация может определять поведение программы лишь на подмножестве множества всех возможных входных данных. Таким образом, не все ошибки могут быть определены при помощи тестирования. Для этого и нужно проводить анализ кода, который позволяет обнаружить такие ошибки, а точнее уязвимые места исходного кода: переполнение буферов, неинициализированная память, указатели (null-pointer), «утечка» памяти, состояние гонок и др. Примеры такого рода ошибок можно увидеть в статье, посвященной статическому или динамическому анализу кода.

Можно сказать, что анализ исходного кода – это процесс получения информации о программе из ее исходного кода или объектного кода. Исходный код – это статическое, текстовое, удобочитаемое, исполняемое описание компьютерной программы, которое может быть скомпилировано автоматически в исполняемый код.

Анализ исходного кода включает три компонента:

- **Парсер (parser)**, который выполняет синтаксический разбор исходного кода и преобразует результаты анализа в одну или несколько форм внутреннего представления. Большинство парсеров (синтаксических анализаторов) основываются на компиляторах.
- **Внутреннее представление**, которое абстрагирует конкретный аспект программы и представляет его в форме, пригодной для выполнения автоматического анализа. Например, переменные заменяются соответствующими типами данных. Некоторые внутренние представления создаются непосредственно парсерами, другие требуют результатов предыдущего анализа. Классическими примерами таких представлений является граф потоков управления (control-flow graph, CFG), дерево вызовов (call graph), абстрактное синтаксическое дерево (abstract syntax tree, AST), форма статического единственного присваивания (static single assignment, SSA).
- **Анализ внутреннего представления**. Анализ может производится по

- Статический или динамический. Статический анализ кода (static code analysis) производится без реального выполнения программ. Результаты такого анализа применимы и одинаковы для всех исполнений программы. В отличие от статического анализа, динамический анализ кода (dynamic code analysis) производится при помощи выполнения программ на реальном или виртуальном процессоре. Результаты такого анализа более точные, но гарантируются только для конкретных входных данных. Утилиты динамического анализа могут требовать загрузки специальных библиотек или даже перекомпиляцию программного кода.
- Контекстно-зависимый (context-sensitive) межпроцедурный анализ.
- Потоково-зависимый (flow-sensitive) анализ.

Кроме того, анализаторы кода можно классифицировать следующим образом.

- Автоматические анализаторы, которые проверяют исходный код в соответствии с предопределенным набором правил и по результатам проверки создают отчеты.
- Различные виды браузеров, которые визуализируют структуру (архитектуру) ПО, таким образом, помогают лучше ее понять.

6.4. Примеры инструментальных систем для отладки

Инструментальные средства Keil Software

Инструментальные средства Keil Software, входящие в пакет uVision, позволяют производить отладку микроконтроллерных систем с помощью симулятора и отладчика. Отладчик умеет подключаться к микроконтроллерам как через интерфейс JTAG (если он есть), так и через RS232 (при этом в микроконтроллере должен быть установлен программный резидентный модуль отладчика).

Основная особенность симулятора состоит в том, что у разработчика существует возможность программирования процесса отладки, создания внешних воздействий и вывода результатов с помощью специального скриптового языка. В остальном, возможности симулятора и отладчика являются совершенно стандартными для настоящего времени:

- отладка в исходных текстах;
- просмотр переменных;
- просмотр стека;
- просмотр регистров и памяти;

- точки остановки.

С помощью программы PONTiFLEX фирмы Adapted Solutions возможно объединение симулятора или отладчика, входящего в пакет uVision с MATLAB/Simulink.

Симулятор электронных схем

Симуляторы электронных схем, такие как Multisim фирмы National Instruments Electronics Workbench Group и Proteus фирмы Labcenter Electronics позволяют не только отлаживать цифровые и аналоговые схемы, но и отлаживать программы совместно с аппаратурой.

В состав программных пакетов входит несколько моделей различных микроконтроллеров, в которые можно загружать программы и исполнять их в непрерывном, пошаговом режиме или до точки остановки.

С помощью виртуальных приборов, таких как вольтметр, амперметр, анализатор I2C, осциллограф, логический анализатор и генератор сигналов можно создать тестовые сигналы и посмотреть результаты работы схемы.

Инструментальные средства отладки ОС РВ eCos

В ОС РВ eCos для отладки ПО используется Redboot. Redboot (акроним от «Red Hat Embedded Debug and Bootstrap») – это приложение с открытым исходным кодом, которое использует слой абстракции оборудования (HAL) операционной системы реального времени eCos для загрузки программного обеспечения или прошивки (firmware) во встроенных вычислительных системах. Redboot предоставляется под GPL-совместимой лицензией eCos License.

Redboot предоставляет широкий набор инструментов для загрузки и исполнения программ в целевых встроенных системах, в том числе Embedded Linux и eCos приложений, через последовательный канал или Ethernet соединение, а также инструменты для манипулирования параметрами целевой системы. Redboot также обеспечивает простую файловую систему для модулей флэш-памяти, которая может быть использована для загрузки исполнимых образов. Он может быть использован при разработке продукции (для поддержки отладки), а также для использования в конечной продукции (для загрузки приложений по сети или с флэш-памяти).

Возможности Redboot:

1. поддержка загрузочных скриптов;
2. простой интерфейс командной строки для управления и конфигурирования Redboot, доступный через последовательный канал или Ethernet соединение по протоколу telnet;

3. встроенные заглушки GDB для соединения с отладчиком на хост-машине через последовательный или сетевой Ethernet интерфейс (ограничивается локальной сетью) для отладки приложений на целевой встроенной системе;
4. атрибутная конфигурация – пользовательский контроль и возможность изменения таких аспектов, как системное время и дата (если используется), статический IP адрес и т.д.;
5. конфигурируемый и расширяемый, специально для адаптации под целевую платформу;
6. поддержка загрузки по сети, включая установку и загрузку через BOOTP, DHCP и TFTP;
7. поддержка загрузки программ через последовательный интерфейс посредством протоколов XModem и YModem;
8. само-тестирование при запуске
9. хотя Redboot является ответвлением от eCos, он может быть использован в качестве общей системы отладки и контроля загрузки программного обеспечения для любых встроенных систем и операционных систем. Например, с соответствующими дополнениями Redboot может заменить широко используемые программы BIOS персональных компьютеров.

Симулятор Armulator

Armulator является симулятором для процессорных ядер ARM. Armulator позволяет симулировать память, регистры, контроллер прерываний, таймер. Одной из основных особенностей Armulator является открытая архитектура, позволяющая самостоятельно реализовывать необходимые модели периферийных модулей.

7. Программирование ВВС с микроэнергопотреблением

Понижение энергопотребления микроконтроллеров является очень важным моментом в процессе проектирования встраиваемых систем. Понижение энергопотребления позволяет увеличить период функционирования контроллеров, работающих от автономных источников электроэнергии (батарей, аккумуляторов), улучшить температурный режим работы для систем с пассивным охлаждением (без вентиляторов охлаждения), работающих в замкнутом объеме (бортовая электроника), упростить схему источника питания. Существует несколько методов уменьшения энергопотребления, которые можно использовать вместе и раздельно, сообразно решаемой задаче:

- отключение питания внешних устройств на плате;
- уменьшение тактовой частоты микроконтроллера;
- режим сна;
- отключение блоков микроконтроллера, которые в данный момент не используются;
- перевод портов ввода-вывода в состояние с минимальным энергопотреблением.

Необходимо заметить, что мало использовать приведенные ниже рекомендации. Они носят общий характер, демонстрируя только базовые принципы снижения энергопотребления. В каждом конкретном случае необходимо подробно изучить схему вашего устройства, а также руководство пользователя и дополнительные замечания по использованию (application notes), поставляемые фирмой производителем микроконтроллера и других электронных компонентов. Кроме того, необходимо помнить, что для достижения нужного эффекта нужно постоянно измерять и записывать энергопотребление платы с помощью мультиметра.

7.1. Отключение питания внешних устройств

Для уменьшения энергопотребления всей схемы можно обеспечить схемотехническую возможность отключения питания для ряда внешних по отношению к микроконтроллеру микросхем. Управление питанием этих схем может осуществляться через выводы микроконтроллера.

7.2. Уменьшение тактовой частоты микроконтроллера

Уменьшение тактовой частоты процессора является одним из самых простых способ уменьшения энергопотребления. В зависимости от типа микроконтроллера может быть достигнуто снижение потребляемой мощности в единицы – десятки раз. Уменьшение энергопотребления происходит из-за

уменьшения числа переключений, производимых транзисторными каскадами в микроконтроллере. Энергопотребление одних схем в большей степени зависит от частоты, других – в меньшей. Поэтому характер зависимости энергопотребления от частоты зависит от типа микроконтроллера, количества включенных в нем блоков, настройки портов ввода-вывода и внешней нагрузки.

Таблица 7. Примерные значения потребляемой мощности для двух различных частот тактового генератора

Микроконтроллер	**f1/P**	**f2/P**
Motorola 68HC705KJ1	1,0 МГц/ 4,0 мВт	2,1 МГц / 4.6 мВт
Microchip PIC 18	32 кГц / 3 мВт	40 Мгц/60 мВт
Philips LPC 9xx (MCS51)	12 МГц/25 мВт	18 МГц/40 мВт
Philips LPC 2292 (ARM7)	10 МГц/13 мВт	60 МГц/90 мВт

В большинстве случаев такой подход не используется, так как при снижении тактовой частоты пропорционально уменьшается производительность микроконтроллера. Более предпочтительным является использование режима сна.

Управление тактовой частотой осуществляется за счет установки соответствующих кварцевых резонаторов. Кроме того, в большинстве современных микроконтроллеров реализованы схемы делителей и умножителей частоты.

7.3. Режим сна

Режим сна является одним из самых мощных способов уменьшения энергопотребления микроконтроллера. В зависимости от типа микроконтроллера и режима сна можно получить выигрыш в энергопотреблении от нескольких раз до нескольких порядков.

В режиме сна тактирование микроконтроллера прекращается, выполнение программы останавливается, энергопотребление становится минимальным. Выход из этого режима возможен в следующих случаях:

- при выключении и последующем включении питания;
- по сигналу RESET, в том числе, вырабатываемым сторожевым (wtachdog) таймером;
- при возникновении не запрещенного прерывания.

В зависимости от типа микроконтроллера, режим сна может отличаться по способу влияния на состояние тактового генератора, регистров, памяти и периферии. В настоящее время для обозначения различных режимов сна в

англоязычной литературе используют термины sleep, standbay, stop, wait и т.д.

Обычно схема работы встраиваемой системы в режиме сна такова:

- при отсутствии каких-либо внешних событий система спит, энергопотребление минимально;
- при возникновении какого-либо события микроконтроллер просыпается, выполняет необходимые действия по обработке события и вновь засыпает.

При использовании сна возможны периодические и апериодические режимы работы. В периодическом режиме микроконтроллер выходит из сна через одинаковые промежутки времени, по прерыванию от таймера. В апериодическом режиме выход из сна происходит по факту прихода внешнего сигнала, к примеру, по факту появления сигнала на дискретном порте ввода вывода, на входе UART или I^2C. Как уже говорилось выше, различные способы уменьшения энергопотребления можно комбинировать. В большинстве случаев, при использовании режима сна нет особого смысла понижать тактовую частоту процессора. При высоком быстродействии задача выполняется быстрее и микроконтроллер раньше засыпает, что приводит к большей экономии электроэнергии. Необходимо заметить, что существует великое множество различных микроконтроллеров, и все они сделаны по разному. Поэтому, в каждом конкретном случае необходимо ставить эксперименты и искать оптимальные соотношения различных способов понижения энергопотребления.

Таблица 8. Примерные значения потребляемой мощности для обычного режима и одного из режимов сна

Микроконтроллер	**Обычный режим**	**Режим сна**
Motorola 68HC705KJ1	4,0 мВт	1.0 мВт
Atmel mega 128	33 мВт	75 мкВт
Microchip PIC	20 мВт (20 МГц)	10 мкВт
TI MSP 430	3 мВт	15 мкВт
STMicroelectronics ARM7	150 мВт (48 МГц)	33 мкВт

7.4. Отключение блоков микроконтроллера

Все знают, что если выключить лишние светильники в комнатах вашей квартиры, то это приведет к снижению энергопотребления. Для того, чтобы не платить много денег за электроэнергию, люди стремятся выключать

неиспользуемое электрооборудование. Аналогичный механизм есть в большинстве современных микроконтроллеров. Микроконтроллер состоит из множества блоков, многие их которых можно включать и выключать (контроллеры UART, CAN, I2C, ЦАП, АЦП, таймеры и т.д.). Если неиспользуемые блоки выключить, то энергопотребление уменьшится.

7.5. Конфигурирование портов ввода-вывода

Необходимо помнить, что от конфигурации портов ввода-вывода зависит не только энергопотребление, но и работоспособность схемы. После сигнала RESET конфигурация всех портов ввода-вывода сбрасывается в исходное состояние. Первое, что нужно сделать – корректно запрограммировать начальное состояние портов в соответствии со схемой вашего устройства и рекомендациями по уменьшению энергопотребления приведенными ниже.

Цифровые порты ввода

Минимальное потребление на цифровых входах микроконтроллера будет в том случае, если напряжение на входе близко к нулю или к напряжению питания. Если напряжение на цифровом входе постоянно меняется или находится между точками ноль-питание, то лучше перевести порт на выход, если это позволяет схема. Необходимо помнить, что если два выхода работающих на вывод соединить вместе, то это может привести к выходу из строя портов. Соединять вместе можно только выходы с открытым коллектором.

Цифровые порты вывода

Потребление цифрового выходного порта обусловлено нагрузкой, которая к нему подключена. Поэтому, если это возможно схемотехнически, для минимизации энергопотребления нужно подать такой сигнал на выход порта, чтобы ток через нагрузку был минимальным.

Аналоговый порт ввода (АЦП)

АЦП имеют высокое входное сопротивление и поэтому потребляемый ток аналоговых входов весьма низок. Минимальное потребление энергии наблюдается в тех случае, когда напряжение входного сигнала находится в середине между нулем и напряжением питания. В ряде случаев выгодно переводить входные порты в аналоговый режим работы для уменьшения энергопотребления.

8. Тестопригодное программирование

8.1. Основные определения

Тестирование – процесс оценки качества системы.

Тестирование – проверка системы на соответствие заданным функциям. Его можно отличать (проводить) на этапе проектирования, производства и эксплуатации. Проверяется функциональность системы (содержательная часть технического задания), соответствие условиям эксплуатации (механическим, климатическим, электромагнитным воздействиям).

Дефект (лат. defectus) – изъян, недостаток. В ГОСТ, термин дефект определяется так: дефект – каждое отдельное несоответствие продукции установленным требованиям.

Дефекты могут иметь постоянный и временный характер.

Сбой – дефект, имеющий временный характер.

Отказ – дефект, имеющий постоянный характер.

Верификация (от лат. verus – истинный, facere – делать) – подтверждение на основе представления объективных свидетельств того, что установленные требования были выполнены. (Определение дано согласно ГОСТ Р ИСО 9000-2001) .

В термин верификация попадают такие понятия, как испытания, исследования, тестирование, рецензирование. Иногда термин верификация путают с валидацией, но это совсем разные вещи.

Верификация – попытка найти ошибки, выполняя программу в тестовой или моделируемой среде [Майерс].

Валидация – это подтверждение на основе представления объективных свидетельств того, что требования, предназначенные для конкретного использования или применения, выполнены. (Определение дано согласно ГОСТ Р ИСО 9000-2001).

Основная **цель валидации** – подтвердить, что предложенное решение подходит для исходной задачи на основе ранее полученных объективных свидетельств. Никакого исследования и тестирования системы не производится.

Цель верификации – удостовериться на практике, что решение задачи удовлетворяет поставленным требованиям.

С точки зрения программного обеспечения, **валидация** – попытка найти

ошибки, выполняя программу в заданной реальной среде [Майерс].

Пример.

Исходная задача: найти сумму 32760 и 23.

Программисту дали требования для более общей задачи: написать программу, находящую сумму двух чисел. В результате этого появился следующий код:

```
#include <stdio.h>

int main(){
        char oneNumber, otherNumber;
        short result;
        oneNumber = otherNumber = result = 0;

        printf("Warning: numbers should be [0..255]\n");
        printf("Number one: ");
        scanf("%c", &oneNumber);
        printf("Number two: ");
        scanf("%c", &otherNumber);

        result = oneNumber + otherNumber;
        printf("%c + %c = %d\n", oneNumber, otherNumber, result);
        return 0;
}
```

Задача, поставленная программисту, решена. Программа проходит верификацию, но валидацию такая программа не пройдёт: исходная задача найти сумму чисел 32760 и 23 осталась нерешенной. Вот как ведет себя программа при попытке сложить нужные числа:

```
Warning: numbers should be [0..255]
Number one: 32760
Number two: 33 + 2 = 101
```

8.2. Общие принципы тестирования

Цели тестирования

Тестирование позволяет ответить на следующие вопросы:

- достаточно ли надежно и безопасно работает система;
- соответствует ли функциональность системы имеющимся спецификациям;
- соответствует ли система требованиям реального времени.

Результатом тестирования является перечень ошибок. Что такое ошибка? К примеру, в энциклопедическом словаре Ефремовой ошибку трактуют так:

1. то, что невозможно рассчитать и предсказать заранее, опираясь на накопленные знания;

2. неправильность в действиях, поступках, суждениях, мыслях.

Другими словами можно сказать, что ошибка это некое внешнее проявление, которые мы замечаем и понимаем что в нашей системе что-то не так. В технике ошибкой обычно считают проявление неисправности [28].

Необходимо заметить, что ошибка и неисправность – совершенно разные вещи.

Неисправность или неисправное состояние – такое состояние объекта, при котором он не соответствует хотя бы одному из требований нормативно-технической и (или) конструкторской (проектной) документации.

Неисправность касается содержания нашей системы, а ошибка только формы. Исходя из этого мы понимаем, что одна и та же неисправность может породить множество ошибок (вспомните, сколько ошибок и предупреждений выдает компилятор Си из-за единственной синтаксической ошибки в вашем исходном тексте). Если говорить медицинскими терминами, то ошибка – это симптом, а неисправность – болезнь, которую нужно лечить. Неявное соответствие ошибки и вызвавшей ее неисправности порождает множество проблем и заставляет с особой тщательностью подходить к построению системы тестирования. Не все тесты позволяют однозначно диагностировать неисправность. Точность, с которой тот или иной тест может локализовать неисправность, называют **разрешающей способностью**.

Для того, чтобы тестирование было не только возможно, но и достаточно эффективно, необходимо разрабатывать систему таким образом, чтобы она была

пригодна для тестирования (DFT, design for test).

Требования к тестируемой системе

1. Разрабатываемая *система* должна проектироваться так, чтобы ее можно было быстро и эффективно протестировать. В данном случае, речь идет не о такой частности, как программное обеспечение, а именно о всей системе в целом.

2. Необходимо знать, в каких местах системы нужно разместить средства для съема показаний, а в каких имитаторы входных воздействий.

3. Необходимо заранее знать, что мы должны увидеть на выходе тестируемой системы при заданном входном воздействии.

4. Необходимо уметь составить такие тестовые наборы, которые обеспечат максимально возможное тестовое покрытие.

Особенности тестирования встроенного ПО

В связи с тем, что встраиваемые системы, как правило, используются для ответственных применений и имеют непосредственный контакт с реальным миром (объектом управления), к тестам предъявляются особые требования.

Особенности тестирования встроенного ПО.

1. По сравнению с ПО систем общего назначения, ПО ВВС имеет гораздо большую сложность из-за того, что вычислительный процесс основан не на одной, а на множестве моделей вычислений, отличных от привычной всем машины Фон-Неймана, предполагающей только последовательное выполнение команд.

2. В ВВС гораздо меньшее количество вычислительных ресурсов по сравнению с системами общего назначения.

3. Распределенность ВВС. Самый распространенный вариант при тестировании ВВС – разделение целевой и инструментальных систем. В более сложных случаях возможна работа с гетерогенной сетью целевых систем со сложной топологией, разнородными узлами сети и различными интерфейсами.

8.3. Тестопригодное проектирование

Для того, чтобы процесс проектирования был тестопригодным, необходимо по возможности стремиться к простоте (принцип KISS), понимать архитектуру системы и стараться проектировать ее так, чтобы в эту систему не только можно было встроить тесты, но и чтобы процесс встраивания тестов был прост и гармоничен. Необходимо также помнить, что в отличие от систем общего назначения, мы имеем дело не с изолированной идеальной системой, а с комплексной, составной системой, включающей в себя объект управления, аппаратное обеспечение, конструктив, целевое и инструментальное программное обеспечение.

Исходя из вышесказанного, становится понятно, что классические средства тестирования, применяемые при создании крупных программных продуктов, могут быть применены лишь частично. Кроме того, подобные средства в нашем случае не решают всех задач. К примеру, разработчику программы для встроенной системы могут заявить следующее: «твоя программа перестает работать при температуре 57 градусов Цельсия» или «программа начинает давать сбои при возникновении вибрации».

Из несвойственных для обычного программирования средств тестирования можно отметить следующие:

- осциллографы и логические анализаторы;
- JTAG;
- программно-аппаратные отладочные стенды (testbench);
- инструментальные машины;
- климатические установки;
- установки для генерации электромагнитных помех;
- вибростенды.

В чем состоит сложность тестирования? К сожалению, тесты создают дополнительные помехи в наблюдаемой системе, внося так называемую инструментальную погрешность. Кроме того, система тестирования, встроенная в целевую систему, «съедает», как правило, очень дорогие ресурсы, которых, как известно, всегда не хватает. Поэтому приходится постоянно искать компромисс между удобством тестирования, влиянием тестов на работу целевой системы и имеющимися ресурсами.

Необходимо понимать, что создание встроенной системы, готовой к тестированию, – сложная, многоаспектная задача, к решению которой нужно подходить очень серьезно.

Рассмотрим минимум, необходимый для создания тестопригодной системы.

Во-первых, необходима какая-либо система регистрации наподобие

самописца. Дело в том, что многие тесты могут длиться по несколько дней, а в таких условиях фиксировать ошибки вручную нет никакой возможности. В качестве системы регистрации удобно использовать инструментальный компьютер с базой данных, хранящей результаты тестирования.

Во-вторых, для тестирования встроенных систем нужен тестовый стенд. Тестовый стенд может быть простым набором проволочек, тумблеров, лампочек и перемычек, а может быть сложным, программно-аппаратным комплексом, снабженным разнообразными имитаторами объекта управления, а возможно и самим объектом управления, если это физически вообще возможно.

В-третьих, для тестирования необходимо уметь создавать неблагоприятную окружающую среду для разрабатываемого устройства. Для этого необходимо проверить, как ведет себя разрабатываемая система в нормальных условиях, в жару, в холод, под воздействием влаги, пыли и вибрации. В простом случае можно обойтись электронагревателем, бытовым холодильником и примитивным генератором помех, а в сложном придется покупать или арендовать перечисленное оборудование в специализированных фирмах, проводящих подобные испытания.

Вернемся к проектируемой системе. Что нужно сделать, чтобы она стала тестопригодной? Ответ в общем-то достаточно прост, если рассматривать систему как комплекс акторных структур, существующих в рамках различных моделей вычислений. Если представить тестируемый элемент системы как актор, то необходимо предусмотреть возможность подачи тестирующего воздействия на вход актора и обеспечить просмотр выходного сигнала на выходе актора.

8.4. Тестирование в диалоговом режиме

Тестирование в диалоговом режиме применяется в тех случаях, когда объем тестирования невелик, или когда система еще до конца не отлажена. Тестовая система должна обязательно иметь в своем составе какой-либо интерфейс пользователя и набор средств для организации тестовых случаев.

Недостатком такого подхода являются:

- наличие человеческого фактора, влияющего на результаты тестирования;
- сравнительно большое время проведения тестов;
- высокая трудоемкость и необходимость в тестировщиках с высокой квалификацией.

К достоинствам подхода можно отнести следующие:

- простота реализации (следуем принципу KISS);

- низкая стоимость реализации;
- гибкость решения.

В связи с тем, что большинство микроконтроллеров имеют в своем составе UART, самым распространенным в среде проектировщиков встроенных систем подходом является реализация системы тестирования через последовательный канал и эмулятор терминала.

В целевой системе для тестирования необходимо иметь:

- последовательный канал RS-232 для связи с инструментальным компьютером;
- примитивный драйвер последовательного канала, работающий по опросу;
- система простейших текстовых меню для вызова тестов и их конфигурирования;
- инструментальный компьютер с последовательным каналом RS-232;
- нуль-модемный кабель RS-232;
- эмулятор терминала (в нашем случае для этого используется программа m3p, входящая в комплект ПО для учебных стендов SDK).

В сравнительно сложных системах, построенных на базе ОС РВ или ОС для встроенных применений (например, embedded Linux), для целей отладки и тестирования используется консоль, доступ к которой организуется как через последовательный канал, так и через Ethernet, с использованием стека протоколов TCP/IP и протоколов telnet или ssh.

8.5. Автоматическое тестирование

Автоматическое (без участия человека) и автоматизированное (с участием человека) тестирование экономически целесообразно, если существует необходимость работы с достаточно большой серией устройств или влияние человеческого фактора оказывает слишком негативный эффект на процесс тестирования.

Сложность автоматической системы тестирования может быть сопоставима со сложностью тестируемой системы. В комплект системы автоматического тестирования должен входить тестовый стенд, оборудованный средствами генерации тестовых воздействий. Кроме того, в тестовой системе должен присутствовать регистратор событий.

Сложность процесса автоматического тестирования многократно возрастает, если тестируемая система является распределенной.

9. Повторное использование при создании ПО ВВС

9.1. Два подхода к организации коллектива разработчиков

Существует два радикально отличающихся друг от друга подхода к организации коллектива разработчиков:

- построение крупной фирмы с большим объемом ресурсов;
- построение компактного коллектива разработчиков.

В первом варианте строится большая фирма, имеющая структуру пирамиды. К сотрудникам в такой фирме предъявляются следующие требования:

- исполнительность;
- умение работать в команде;
- средний уровень знаний;
- умение пользоваться текущей версия инструментария.

Работа фирмы характеризуется большим количеством бюрократических процедур и сравнительно низкой производительностью. За счет большого количества взаимозаменяемых сотрудников фирмы такого рода отличаются большой стабильностью результатов и в состоянии решать огромные по сложности задачи. По такому принципу строится большинство современных западных фирм, на такие фирмы настроено современное западное образование.

Второй подход предполагает создание низкобюджетной, компактной фирмы с небольшим количеством разработчиков. К сотрудникам предъявляются следующие требования:

- наличие критического мышления;
- умение думать самостоятельно и нестандартно;
- умение учиться самостоятельно;
- концептуальное, абстрактное мышление;
- фундаментальное образование.

Плюсом в такой организации является высокая производительность сотрудников, в одиночку способных заменить целый отдел обычных разработчиков (см. книгу Брукса «Мифический человеко-месяц»). К минусам можно отнести более низкую стабильность результатов, плохую взаимозаменяемость сотрудников. Людьми в такой фирме достаточно сложно управлять. Кандидатов для приема на работу в такой фирме значительно меньше.

Первый тип фирм, как правило, занимается большими и гигантскими проектами. Об организации работ в таких фирмах написано достаточно много литературы и мы не будем на них останавливаться.

Второй тип фирм занимается низко- и среднебюджетными проектами с большой долей научно-исследовательской составляющей. После развала СССР на территории России организовалось большое количество небольших фирм второго рода, начинающих свое развитие с нуля. Вот как раз о таких фирмах и пойдет у нас речь.

9.2. Проблемы повторного использования

Без применения каких-либо методик повторного использования распространение заготовок за пределы рабочей группы практически невозможно.

Между рабочими группами всегда существует некий «потенциальный барьер», препятствующий распространению полезной информации. Барьер формируется, как правило, из-за нежелания сотрудничать, различных уровней подготовки сотрудников, разных стилей руководства, отличных друг от друга подходов к проектированию и т.д. Проникновение информации через барьер возможно только при наличии достаточно большой энергии, прикладываемой с обеих сторон для организации совместной деятельности.

Практика показывает, что при определенной сложности компонента сделать такой-же значительно проще с нуля, чем перенять чужой опыт. Почему это происходит?

- Исходные тексты могут ответить на вопрос *«как это сделано»*, но они никогда не ответят на вопрос *«почему это именно так сделано»*.

- Сколько-нибудь сложная система требует наличия проектной документации, чтобы в ней можно было разобраться и поддержки со стороны разработчиков, чтобы можно было узнать то, что непонятно.

- Необходимо понимание общей концепции системы для того, чтобы можно было понять частности.

К сожалению, в низкобюджетных разработках не делают сколько-нибудь серьезной проектной документации из-за недостатка времени, исходные тексты, как правило, делаются без комментариев, стиль кодирования оставляет желать лучшего. Информация о концепции не изложена на бумаге, а хранится только в одном месте – в голове ведущего разработчика или архитектора проекта. Общая занятость коллектива не позволяет тратить время на объяснения тонкости работы какого-либо компонента другой рабочей группе.

Перечисленные проблемы решаются, если увеличить бюджет проекта. В ряде случаев такое экстенсивное решение может сделать проект нерентабельным. К примеру, требуется сделать уникальное устройство со сравнительно небольшим тиражом и небольшим бюджетом разработки. Выполнение такой работы крупной фирмой невозможно из-за плохой рентабельности, а мелкая фирма может не справиться с работой из-за проблем с повторным использованием. Как решить эту проблему?

Можно предложить два способа.

- Повторное использование на архитектурном уровне.
- Повторное использование в рамках базовой архитектуры.

9.3. Повторное использование на архитектурном уровне

Повторное использование на уровне архитектуры предполагает хранение и передачу информации в виде идей и концепций, а не в виде реализации.

Достоинством такого подхода является минимально возможный объем передаваемой информации.

Недостатком такого подхода является большая сложность передаваемой информации и, как следствие, необходимость владения системным видением проблем. К сожалению, не все разработчики в состоянии размышлять и рассуждать на концептуальном уровне. О низком уровне подготовки специалистов говорит уровень справочной технической литературы. Книг, дающих концептуальный взгляд на вещи, практически нет, так как требуется высокая квалификация для их написания и не менее высокая квалификация для их верного понимания.

9.4. Повторное использование в рамках базовой архитектуры

Суть повторного использования на уровне реализации в рамках базовой архитектуры заключается в использовании шаблона, ограничивающего спектр проектных решений разных рабочих групп и направляющего эти группы в примерно одинаковом направлении. В качестве такого шаблона может выступать операционная система реального времени, библиотека, входящая в состав инструментальных средств программирования, спецификация на сетевую подсистему, например, такая как CanOpen или DeviceNet.

Для небольшой фирмы, как правило, нерентабельно создавать свои собственные, сколько-нибудь сложные базовые архитектуры. Базовые архитектуры без должного сопровождения и документирования едва когда-нибудь выйдут за пределы создавшей их рабочей группы.

Повторное использование на уровне исходных текстов

Внешне этот способ повторного использования выглядит самым простым. На самом деле это не совсем так, по следующим причинам.

- В силу причин, изложенных в главах о стиле программирования и моделях вычислений, исходные тексты не несут (и не могут нести в принципе) в себе высокоуровневой информации об архитектуре. Из этого следует, что они непонятны для сторонних разработчиков, не имеющих тесного контакта с разработчиком архитектуры. По сути, сколько-нибудь сложный исходный текст очень часто оказывается черным ящиком.
- Исходный текст очень просто повредить или незаметно для остальных участников проекта исправить, внеся в проект несколько иную функциональность или ошибки. Кроме того, внося изменения в исходный текст, мы можем породить большое количество новых ветвей проекта, в которых очень легко запутаться.
- Всегда существуют проблемы с созданием инструментальной среды, необходимой для сборки текущей версии исходных текстов. В относительно стабильной среде систем общего назначения, использующих очень широко распространенные инструментальные средства крупных корпораций, эта проблема не стоит так остро. В области же встроенных систем, разнообразие инструментальных средств гораздо выше и существует достаточно серьезная проблема зависимостей. Суть проблемы заключается в том, что для успешной сборки проекта вам нужно собрать в одно время и в одном месте большое количество компонентов. Попробуйте, к примеру, собрать из исходных текстов компилятор GCC для процессора ARM и операционную систему реального времени eCos под Linux и вы поймете, что проблема реально существует.

Исходя из вышесказанного, можно предложить несколько частных решений, позволяющих частично компенсировать изложенные проблемы.

- Для компенсации сложности исходных текстов необходимо применять следующие методы.
  - Организовать по возможности некий единый стиль кодирования. Как минимум, это позволит сделать исходные тексты лучше читаемыми.
  - Убедить разработчиков использовать идиоматичные конструкции, которые будут порождать у тех, кто читает исходные тексты ассоциации с готовыми, шаблонными архитектурными решениями.
- Проблемы с намеренным или случайным повреждением исходных текстов, а также борьба с версиями исходных текстов может быть решена за счет грамотной организации репозитория исходных текстов. Как минимум, этот репозиторий должен иметь архивные копии. Желательно, чтобы сохранение резервной копии репозитория производилось периодически, по мере достижения каких-либо существенных результатов. Это позволит произвести откат на предыдущие шаги, в случае появления трудно уловимых ошибок и каких либо других проблем. Более комплексно эта проблема решается при использовании системы контроля версий вроде CVS или Subversion. В использовании инструментальных средств для контроля за исходными текстами нужно знать меру, понимая, что ресурсы компактной рабочей группы далеко не безграничны и сложное инструментальное средство либо не будет эффективно в использовании, либо ни кем не будет использовано вообще.
- Проблема с созданием инструментальной среды, адекватной имеющимся исходным текстам, на данный момент весьма серьезна. Попытки создания автоматических конфигураторов библиотек, поиска зависимостей при установке библиотек сейчас пытаются предпринять многие разработчики инструментальных систем. К сожалению, эти решения касаются в основном систем общего назначения, а не встроенных систем. В качестве удачных примеров решения проблемы можно привести ряд инструментальных средств, распространенных в Unix подобных операционных системах: скрипт .configure для адаптации программного проекта к имеющимся библиотекам, репозиторий дистрибутивов инструментальных средств и библиотек с автоматическим поиском зависимостей и разрешений конфликтов, с возможностью получения необходимых компонентов через Интернет (в качестве примера такого инструментального средства можно привести программу Yast, входящую в состав SUSE Linux. Необходимо заметить, что создание подобных средств силами малых коллективов весьма маловероятно в силу большой трудоемкости.

Повторное использование на уровне объектных файлов

Использование объектных файлов решает проблему сложности исходных текстов, так как они не доступны разработчику. Кроме того, дополнительным плюсом библиотек является невозможность внесения изменений и ошибок в объектные файлы без наличия исходных текстов. Проблема конфигурирования инструментальных средств, необходимых для сборки проекта, остается такой-же острой, как и в случае использования исходных текстов.

Отрицательным моментом в использовании библиотек является невозможность использования наработок для различных аппаратных платформ. К сожалению, в области встроенных систем вариации в реализации аппаратного обеспечения очень сильны.

Еще одним отрицательным моментом является то, что при изменении решаемой задачи старая библиотека может вести себя неадекватно. В лучшем случае, система, собранная из старых библиотек, не пройдет валидацию, в худшем – мы получим эффект, схожий с тем, что был получен при неудачном пуске французской ракеты Ариан-5, в которой использовали старый «проверенный» библиотечный элемент, не подходящий для нового варианта использования.

Повторное использование на уровне компонентов

Использование готовых компонентов частично решает проблему инструментария, т.к. объем требуемых требуемых инструментальных средств существенно ниже. С точки зрения опасности получения неожиданных свойств, проверенных в прошлых проектах компонентов, ситуация аналогична той, что существует при повторном использовании на уровне библиотек.

В целом, это наиболее эффективный способ повторного использования для малых коллективов разработчиков. Какие варианты повторного использования на уровне компонентов возможны?

- В системах с небольшим количеством вычислительных ресурсов, к примеру, в системах на базе микроконтроллеров PIC и AVR младших моделей, минимальным компонентом который можно будет выделить контроллер с программным обеспечением в виде firmware.
- В системах с достаточно большим количеством ресурсов компонентом может стать контроллер с заранее прошитой базовой системой ввода-вывода, операционной системой реального времени или виртуальной машиной.
- В системах с большим объемом ресурсов, работающих под управлением крупных ОС типа QNX или EmbeddedLinux, возможно выделение таких компонентов, как программа.

Приложение 1. Программа для настройки PLL микроконтроллера из сеймейства FR50 фирмы Fujitsu.

```
void config_pll( PLLReg reg )
{
    BYTE  reg_icr[48];
    BYTE  reg_tbcr;
    BYTE* ptr;
    BYTE  i;

#pragma asm
    //сохраняем контекст в стеке
    ST   PS, @-R15
#pragma endasm

    __DI();

    //сохраняем приоритеты прерываний и устанавливаем
    //минимальный приоритет
    for (i = 0, ptr = (BYTE*)0x000440; i < 48; i++)
    {
        reg_icr[i] = *ptr;
        *ptr++ = 31;
    }

    //устанавливаем приоритет прерывания от TBC не
    //равным 0x1F для выхода из sleep-режима
    ICR31 = 30;

    reg_tbcr = TBCR;


    //устанавливаем CLKB, CLKP и CLKT на PLL/16
    //понижая частоту на шинах
    DIVR0 = 0xFF;
    DIVR1 = 0xF0;

    //если генератор тактовых импульсов настроен
    //на использование PLL
    if ((CLKR & 0x03) == 0x02)
    {
        //настраиваем TBC
        CTBR  = 0xA5;
        CTBR  = 0x5A;
        TBCR  = 0x40;

        //переходим в sleep-режим и переключаем генератор
        //на использование осциллятора
        STCR  = 0x50;
        CLKR &= 0xFC;

        while (!TBCR_TBIF);
    }

    //выключаем PLL и устанавливаем 1 wait-state для
    //внутреней flash-памяти
    CLKR &= 0xF8;
    FMWT  = 0x01;

    //если требуется умножение PLL
```

```
    if (reg.clkr & 0x70)
    {
        CTBR  = 0xA5;
        CTBR  = 0x5A;
        TBCR  = 0x00;

        //устанавливаем значения умножителя PLL и wait-state
        //для внутренней flash-памяти
        CLKR &= 0x8F;
        CLKR |= reg.clkr & 0x70;
        FMWT  = reg.fmwt;

        //включаем PLL
        CLKR |= 0x04;

        while (!TBCR_TBIF);

        CTBR  = 0xA5;
        CTBR  = 0x5A;
        TBCR  = 0x40;

        //настраиваем генератор тактовых импульсов на
        //использование PLL и переходим в sleep-режим
        STCR  = 0x50;
        CLKR |= 0x02;

        while (!TBCR_TBIF);

        //выключаем TBC
        TBCR  = 0x00;
    }

    DIVR0 = reg.divr0;
    DIVR1 = reg.divr1;

    //восстанавливаем приоритеты прерываний
    for (i = 0, ptr = (BYTE*)0x000440; i < 48; i++)
    {
        *ptr = reg_icr[i];
        ptr++;
    }

    TBCR = reg_tbcr;

#pragma asm
    //восстанавливаем контекст
    LD  @R15+, PS
#pragma endasm

}
```

Приложение 2. Пример программы для записи внутренней FLASH памяти для микроконтроллеров PIC 16 фирмы Microchip

```
char WriteBuff(unsigned char n)
{
unsigned char tmpH, tmpL;
unsigned char count,retry;

    tmpaddr = address;
    retry   = MaxRetry; // Количество попыток записи во FLASH
    count   = 0;
    while (count < n)
    {
        if( (tmpaddr >= LoaderLOW) && (tmpaddr < LoaderHI) )
            return FLASH_ADDR;
            // Ошибка! Попытка стереть загрузчик
        EEADR=tmpaddr;
        if (tmpaddr < 4)
            EEADRH=LoaderStart>>8;    // Переадресация
        else
            EEADRH=tmpaddr >> 8;
        EEDATH = tmpH = buff[count+1];
        EEDATA = tmpL = buff[count];
        EEPGD  = 1;
        WREN=1;
        EECON2 = 0x55;     // Последовательность для записи
        EECON2 = 0xaa;
        WR=1;
        asm("nop");
        asm("nop");       // Задержка, ждем окончания записи
        WREN=0;
        EEPGD=1;
        RD=1;             // Читаем для верификации записи
        asm("nop");       // Задержка, ждем окончания чтения
        asm("nop");
        if ((EEDATH != tmpH) || (EEDATA != tmpL))
        {
          if (--retry)
            continue;   // Неудачная попытка, пробуем еще раз
          else
            return FLASH_ERROR;  // Попытки записи исчерпаны,
                        // ошибка
          }

        count+=2;         // Запись успешна, переходим к
        ++tmpaddr;       // другому адресу
    }
    return FLASH_OK;     // Успешное завершение
}
```

Литература

1. Керниган, Пайк. Unix. Программное окружение. – СПб.: Символ-Плюс, 2003. – 416 с.
2. Керниган, Ритчи. Язык программирования Си. – СПб.: Невский Диалект, 2000. – 352 с.
3. Ключев А.О. Методы и инструментальное обеспечение разработки распределенных информационно-управляющих систем с программируемой архитектурой: дис... канд. техн. Наук: 05.13.13. – Санкт-Петербург, 1998.
4. Линус Торвальдс, Дэвид Даймонд. Just for FUN. Рассказ нечаянного революционера. – СПб.: Эксмо-Пресс, 2002. – 288 с.
5. Лукичев А.Н. Денотативно-объектная модель вычислений для встроенных систем: дис... канд. техн. Наук: 05.13.12. – Санкт-Петербург, 2008.
6. Вирт Н. Долой "жирные" программы // МИР ПК–ДИСК. 2005. №9 (Источник: Открытые системы, #06/1996).
7. Непейвода Н. Н. Неформализуемость как логическая характеристика жизни // Online Journal "Logical Studies" No.3, 1999 г.
8. Непейвода Н. Н. Стили программирования как общий подход к системе понятий информатики. Режим доступа: http://is.ifmo.ru/aboutus/_log_prog2.pdf
9. Непейвода Н. Н. Стили и методы программирования. Режим доступа: http://www.intuit.ru/department/se/progstyles/
10. Непейвода Н. Н., Скопин И. Н. Основания программирования. – Ижевск–Москва: РХД, 2003. – 868 с.
11. Фредерик Брукс. Мифический человеко-месяц, или Как создаются программные системы. – СПб.: Символ-Плюс, 2007. – 304 с.
12. Шагурин И.И. Архитектура и функционирование микроконтроллеров семейства Motorola 68HC705// Chip News. – 1999. № 3. – С. 2–9.
13. Реймонд Э. Искусство программирования для Unix.
14. Ecos. Режим доступа: http://ecos.sourceware.org/
15. Edward A. Lee, "The Problem with Threads," in IEEE Computer, 39(5):33-42, May 2006 as well as an EECS Technical Report, UCB/EECS-2006-1, January 2006.
16. Embedded Development Tools Keil Software. Режим доступа:

http://www.keil.com/
17. Gerard J. Holzmann. The Power of 10: Rules for Developing Safety-Critical Code NASA/JPL Laboratory for Reliable Software.
18. GNU ARM toolchain Режим доступа: http://www.gnuarm.org/
19. Microchip PIC18FXX8 Data Sheet.
20. Philips LPC2292/LPC2294 Rev. 03 – 1 November 2005 Product data sheet.
21. Philips P89LPC915/916/917 Rev. 04 – 17 December 2004 Product data sheet.
22. SDCC – Small Device C Compiler. Режим доступа: http://sdcc.sourceforge.net/
23. Википедия – свободная энциклопедия. Режим доступа: http://ru.wikipedia.org
24. Микроконтроллеры ARM7/ARM9 обзор. Режим доступа: http://www.phyton.ru/pages/page41.html#energ
25. Советы по уменьшению энергопотребления микроконтроллеров. Режим доступа: http://www.compel.ru/catalog/microcontrollers/microchip/articles/a006
26. Anthony J. Massa Embedded Software Development with eCosT Prentice Hall PTR November 25, 2002 ISBN : 0-13-035473-2.
27. Hatley D.J., Pirbhai I.A. Strategies for Real-Time System Specification. - N.Y. Dorset House Publishing, 1988.
28. Микропроцессоры. Т3. Средства отладки, лабораторный практикум и задачник. под ред. Преснухина.

СПбГУ ИТМО стал победителем конкурса инновационных образовательных программ вузов России на 2007–2008 годы и успешно реализовал инновационную образовательную программу «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий», что позволило выйти на качественно новый уровень подготовки выпускников и удовлетворять возрастающий спрос на специалистов в информационной, оптической и других высокотехнологичных отраслях науки. Реализация этой программы создала основу формирования программы дальнейшего развития вуза до 2015 года, включая внедрение современной модели образования.

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра ВТ СПбГУ ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России.

Первоначально кафедра называлась кафедрой математических и счетно-решающих приборов и устройств и занималась разработкой электромеханических вычислительных устройств и приборов управления. Свое нынешнее название кафедра получила в 1963 году.

Кафедра вычислительной техники является одной из крупнейших в университете, на которой работают высококвалифицированные специалисты, в том числе 8 профессоров и 16 доцентов, обучающие около 500 студентов и 30 аспирантов.

Кафедра имеет 6 компьютерных классов, объединяющих более 70 компьютеров в локальную вычислительную сеть кафедры и обеспечивающих доступ студентов ко всем информационным ресурсам кафедры и выход в Интернет. Кроме того, на кафедре имеются учебные и научно-исследовательские лаборатории по вычислительной технике, в которых работают студенты и аспиранты кафедры**.**

В 2007-2008 гг. коллективом кафедры была успешно реализована инновационная образовательная программа СПбГУ ИТМО по научно-образовательному направлению №2 «Встроенные вычислительные системы».

Ключев А.О., Кустарев П.В., Ковязина Д.Р., Петров Е.В.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВСТРОЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

УЧЕБНОЕ ПОСОБИЕ

Санкт-Петербург

2009

Редакционно-издательский отдел
Санкт-Петербургского государственного
университета информационных
технологий, механики и оптики
197101, Санкт-Петербург, Кронверкский пр., 49

Аркадий Олегович Ключев
Павел Валерьевич Кустарев
Динара Раисовна Ковязина
Евгений Владимирович Петров

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВСТРОЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Учебное пособие

В авторской редакции
Дизайн И.И. Иванов
Верстка П.П. Петров
Редакционно-издательский отдел Санкт-Петербургского государственного университета информационных технологий, механики и оптики
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати 04.06.2009
Заказ № 2119
Тираж 100 экз.
Отпечатано на ризографе