Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Пермский национальный исследовательский политехнический университет»

Кавалеров М.В.

Системное программное обеспечение управляющих систем реального времени

Учебное пособие

Пермь 2013

Системное программное обеспечение управляющих систем реального времени: учеб. пособие / М.В. Кавалеров. – Пермь: Изд-во Перм. нац. иссл. политех. ун-та, 2013. – 190 с.

В учебном пособии изложены базовые понятия, относящиеся к: системному программному обеспечению; управляющим системам реального времени; планированию задач реального времени. Также рассмотрены: основы работы с интегрированной средой разработки Code::Blocks; пример разработки программ, использующих потоки POSIX; пример разработки прототипа драйвера применительно к ядру Linux. Приведен обзор современного состояния научных исследований, связанных с планированием задач реального времени в системах управления. Пособие является учебным материалом для дисциплины «Системное программное обеспечение управляющих систем реального времени». Ориентировано на студентов магистерской подготовки, а также аспирантов, специализирующихся в области управляющих систем реального времени.

© ГОУ ВПО ПНИПУ, 2013

Содержание

Предисловие

Дисциплина «Системное программное обеспечение управляющих систем реального времени» является частью сетевой магистерской программы 22040056.68 «Информационные технологии в проектировании управляющих систем реального времени» по направлению 220400 «Управление в технических системах».

Сетевой характер программы обусловлен стремлением к повышению качества подготовки за счет объединения усилий (материальной базы и квалификации специалистов) нескольких учебных заведений. Участвующие в сетевой программе университеты ориентируются на разные виды системного программного обеспечения управляющих систем реального времени и используют различные средства разработки программного обеспечения. Теоретические основы курса являются фундаментом, необходимым для понимания общих принципов организации разных видов системного программного обеспечения управляющих систем реального времени, а также общих подходов к разработке и использованию системного программного обеспечения управляющих систем реального времени с применением различных инструментальных средств.

Трудоемкость дисциплины «Системное программное обеспечение управляющих систем реального времени» составляет 180 часов, из них лекции – 6 часов, практические занятия – 18 часов, лабораторные занятия – 16 часов и самостоятельная работа – 100 часов.

Цель учебной дисциплины состоит в ознакомлении студентов с основами разработки и использования системного программного обеспечения управляющих систем реального времени с применением современных инструментальных средств.

Учебное пособие призвано помочь в формировании следующих дисциплинарных профессиональных специальных компетенций магистрантов:

– способность применять средства системного программного обеспечения в процессе разработки управляющих систем реального времени;

– способность применять современные инструментальные средства для разработки и использования системного программного обеспечения управляющих систем реального времени.

Учебное пособие разбито на два раздела. Первый раздел посвящен общим вопросам организации системного программного обеспечения управляющих систем реального времени, а второй – вопросам, связанным с разработкой и использованием системного программного обеспечения управляющих систем реального времени с применением современных инструментальных средств. После каждого раздела приведены вопросы для самоконтроля.

Список сокращений

ЖРВ – жесткое реальное время.

МРВ – мягкое реальное время.

ПЛИС – программируемая логическая интегральная схема.

ПО – программное обеспечение.

ПФП – планирование с фиксированными приоритетами.

РВ – реальное время (реального времени).

СРВ – система реального времени.

1. Основы организации системного программного обеспечения управляющих систем реального времени

1.1. Системное программное обеспечение и управляющие системы реального времени

1.1.1. Вычислительное устройство, аппаратное и программное обеспечение

Для начала обратимся к некоторым базовым понятиям.

Вычислительное устройство – техническое устройство, которое способно выполнять вычисление согласно заданному алгоритму на основе входных данных и формировать результат вычисления на основе выходных данных этого алгоритма.

В простейшем случае алгоритм может состоять из одной или нескольких математических операций, и тогда вычислительным устройством можно считать простой логический элемент (например, «И», «ИЛИ», «И-НЕ») или комбинационное логическое устройство (например, сумматор, компаратор, шифратор).

В более сложных случаях вычислительное устройство может быть представлено совокупностью многих логических элементов, обладать памятью, выполнять сложные алгоритмы.

В качестве примеров сложных вычислительных устройств можно привести микропроцессоры, однокристальные микроконтроллеры.

При подключении к вычислительному устройству внешней памяти, средств ввода-вывода, другого дополнительного оборудования можно говорить о построении *вычислительной системы* (системы взаимодействующих компонентов для выполнения вычислений). Примерами вычислительных систем являются: компьютеры, программируемые логические кон-

контроллеры, мобильные телефоны, смартфоны, суперкомпьютеры.

Надо отметить, что сейчас трудно провести границу между вычислительным устройством и вычислительной системой. Например, в случае системы на кристалле (system on a chip) мы имеем дело с одной микросхемой, обладающей функциональностью, схожей с функциональностью материнской платы персонального компьютера. Поэтому в дальнейшем мы не будем делать каких-либо жестких различий между вычислительным устройством и вычислительной системой.

Электронные и механические части вычислительного устройства или системы образуют *аппаратное обеспечение*.

Часто возникает потребность в изменении алгоритма вычислений.

Алгоритм, выполняемый вычислительным устройством или системой, можно изменить на основе изменения аппаратного обеспечения, например, за счет замены, добавления тех или иных логических элементов или за счет изменения связей между ними. Этот способ имеет свои очевидные плюсы и минусы.

Появление программируемых вычислительных устройств дало возможность изменять алгоритм вычисления без изменения аппаратного обеспечения. На вход программируемого вычислительного устройства можно подавать данные особого рода, представляющие собой *программу*, которую можно рассматривать как описание алгоритма вычисления, который должен быть выполнен на данном устройстве. Это может выполняться в особом режиме функционирования этого устройства, в режиме его программирования. По окончанию программирования, данное устройство переводится в обычный режим функционирования и может начинать выполнять заложенную в него новую программу (новый алгоритм вычисления). В случае сложных вычислительных устройств и систем бывает сложно разделить режим программирования и режим обычного функционирования, например, в случае интерактивного взаимодействия с человеком, когда он

может менять параметры программы, изменять алгоритм вычислений без остановки работы основной программы.

Совокупность программ, заложенных в вычислительное устройство или систему, образуют *программное обеспечение*.

Несмотря на интуитивно понимаемое различие между аппаратным и программным обеспечением, иногда бывает сложно провести между ними границу. Например, в случае использования программируемой логической интегральной схемы (ПЛИС) в качестве вычислительного устройства возникает вопрос. Считать ли изменение логической конфигурации ПЛИС ее программированием, или это всего лишь изменение ее аппаратной конфигурации, то есть, по сути, изменение аппаратного, а не программного обеспечения? Ответ на этот вопрос, скорее всего, определяется тем, насколько часто и при каких условиях осуществляется это программирование. Так, если программирование ПЛИС выполняется один раз и в дальнейшем в ходе работы системы не предполагается ее новое программирование, то удобнее считать текущую логическую конфигурацию ПЛИС особенностью аппаратного обеспечения системы. Если же в ходе эксплуатации системы предусматривается возможность перепрограммирования ПЛИС согласно заданным инструкциям, то тогда файл с текущей конфигурацией ПЛИС можно отнести к программному обеспечению системы.

1.1.2. Системное и прикладное программное обеспечение

Программное обеспечение сложных вычислительных устройств и систем может занимать большой объем. При этом существенную часть этого объема отводится для реализации взаимодействия с типовым аппаратным обеспечением, для выполнения функций, которые характерны для различных вычислительных устройств и систем.

Например, хранение данных в долговременной памяти удобно выполнять с помощью файлов на основе той или иной файловой системы. В

этом случае отдельные блоки данных становятся доступны в виде файлов – логически единых блоков данных, хотя на самом деле физически эти данные могут размещаться в разных местах запоминающего устройства. Можно было бы для каждой новой вычислительной системы создавать новое программное обеспечение, которое реализует файловую систему. Но в большинстве случаев в этом нет необходимости, и лучше использовать готовые реализации файловых систем, которые уже раньше были многократно опробованы и протестированы.

Аналогично этому, во многих случаях для обеспечения работы с типовым периферийным оборудованием также лучше использовать готовые и проверенные компоненты, чем «изобретать заново велосипед».

При таком подходе в составе программного обеспечения будут применяться компоненты, которые были разработаны ранее и которые реализуют типовые функции, необходимые для работы вычислительной системы, то есть можно говорить о том, что они реализуют *системные* функции. В итоге, мы приходим к понятию системного программного обеспечения.

Системное программное обеспечение (ПО) – это совокупность программ, которые обеспечивают взаимодействие с различными компонентами вычислительной системы (процессорами, запоминающими устройствами, устройствами ввода-вывода и др.) и которые реализуют типовые функции, необходимые для работы с этими компонентами.

Системное ПО, как правило, разрабатывается не для отдельной вычислительной системы, а для множества однотипных или схожих вычислительных систем. Такой подход возможен именно в силу того, что системное ПО реализует типовые функции, которые нужны и применимы во многих случаях, например: поддержка файловой системы, управление загрузкой программ пользователя, реализация взаимодействия с типовыми устройствами ввода-вывода.

Но однотипные вычислительные системы, даже несмотря на то, что

они могут использовать одно и то же системное ПО, часто выполняют различные задачи за счет выполнения различных *приложений* – программ, создаваемых для решения конкретных *прикладных* задач. Совокупность этих приложений формирует *прикладное ПО* данной вычислительной системы.

Таким образом, ПО вычислительной системы можно разделить на системное и прикладное.

Системное ПО является своеобразной прослойкой, интерфейсом между аппаратным обеспечением и прикладным ПО. Системное ПО во многих случаях позволяет «скрыть» от прикладного ПО детали аппаратной реализации вычислительной системы, беря на себя многие рутинные типовые операции, что дает возможность создателю прикладного ПО сосредоточиться на решении своей конкретной практической задачи.

1.1.3. Основные виды системного программного обеспечения

Основные виды системного ПО соответствуют основным типовым функциям, необходимым для работы с компонентами вычислительной системы. Можно выделить следующие основные виды системного ПО:

– операционные системы;

– загрузчики;

– драйверы;

– вспомогательные программы (утилиты, командные оболочки и др.);

– системы программирования.

Также иногда к системному ПО относят так называемое *промежуточное* ПО (middleware), которое обеспечивает взаимодействие между различными компонентами программного обеспечения. Часто промежуточное ПО рассматривают как дополнительный слой между системным ПО и прикладным ПО. Во многих случаях для простоты можно считать промежуточное ПО одной из разновидностью системного ПО, так как промежуточ-

ное ПО, также как и системное ПО, выполняет некоторые типовые функции, в частности, обеспечивает связь различных программных компонентов, например, обеспечивая единые программные интерфейсы.

1.1.4. Ограничения реального времени

Система управления взаимодействует с объектом управления на основе информации, поступающей от различных датчиков. Требуется, чтобы модель состояния объекта, формируемая для себя системой управления, была адекватной текущему состоянию данного объекта. Иначе воздействие системы управления на этот объект может оказаться вредным или даже разрушительным. Поэтому система управления должна периодически отслеживать текущее состояние объекта управления и обрабатывать новую информацию для формирования возможных управляющий воздействий.

Требования своевременности получения, обработки информации и формирования управляющий воздействий возникают из-за физического (реального) взаимодействия системы управления и объекта управления. Другими словами, *реальная* природа взаимодействия системы управления и объекта управления порождает требования *реального* времени (РВ), то есть своевременности функционирования системы управления. Например, если сборочному роботу не будет своевременно выдана команда об остановке или начале нового действия, то это может привести к серьезным повреждениям объекта сборки.

В процессе проектирования системы управления эти требования учитываются в виде формально и численно определенных *ограничений* РВ.

Примеры ограничений РВ:

– показания датчика должны считываться каждые 10 мс с допустимой погрешностью во времени в 1 мс;

– команда на открытие клапана должна быть сформирована не позднее чем, через 5 мс после поступления аварийного сигнала.

1.1.5. Понятие системы реального времени

Вычислительная система реального времени (СРВ) – это такая вычислительная система, у которой правильность функционирования зависит не только от логического результата вычислений, но и от физического времени, когда эти результаты формируются [53].

Другими словами, правильность функционирования вычислительной СРВ зависит не только числовых значений формируемых результатов, но и от того, соблюдаются ли ограничения РВ, относящиеся к этим результатам. Предполагается, что ограничения РВ описывают правильность функционирования системы по отношению ко времени.

В определении говорится о *вычислительной* системе, то есть системе, в которой выполняются вычислительные процессы, например, при помощи микропроцессорных устройств. И такую систему можно рассматривать в виде «черного ящика», на вход которого поступают некоторые данные, например, данные с датчиков, а на выходе формируются управляющие воздействия. При этом вычислительная система РВ может взаимодействовать с *объектом управления* (рис. 1). В такой ситуации вычислительная СРВ представляет собой систему управления, реализованную на основе тех или иных вычислительных устройств.

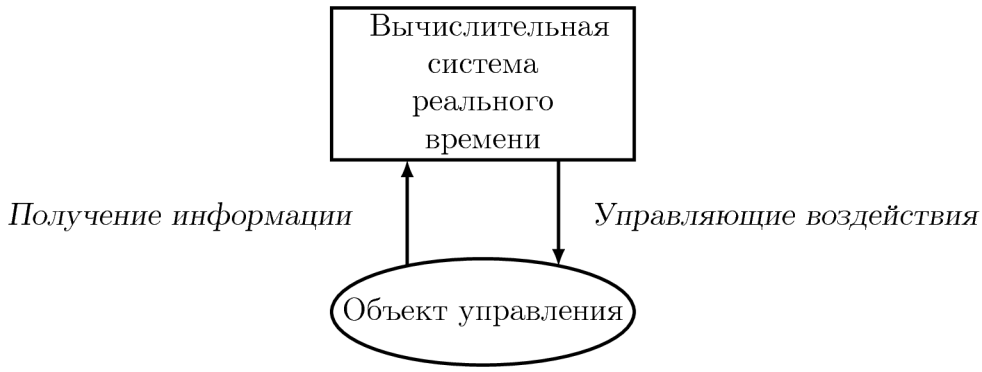

Рис. 1. Вычислительная система реального времени и объект управления

Естественно, что вычислительная СРВ также может взаимодействовать и с *человеком-оператором*, получая от него команды и предоставляя

ему необходимую информацию, и в этом случае объекта управления, как такового, может и не быть (рис. 2). Примерами подобных систем являются виртуальные тренажеры, а также динамические компьютерные игры. В случае указанных примеров на основе вычислительной СРВ создается виртуальная реальность, с которой взаимодействует человек. Эта виртуальная реальность может быть имитацией окружающего нас мира с известными нам физическими законами или имитацией некоторого вымышленного мира компьютерной игры. В любом случае, вычислительная система должна реагировать на действия человека своевременно (без неоправданных задержек), иначе имитация не будет достигнута, и такая система не будет признана правильно функционирующей. Другими словами, для реализации виртуальной реальности требуется вычислительная СРВ.

Рис. 2. Вычислительная система реального времени и человек-оператор

При обобщенном подходе, удобнее рассматривать комплексную систему, в которой вычислительная СРВ взаимодействует как с объектом управления, так и с человеком-оператором (рис. 3). Такую комплексную систему можно называть *комплексной СРВ*.

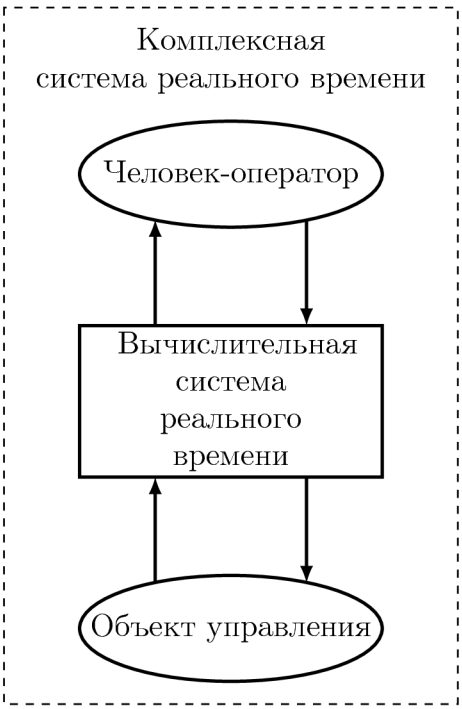

Рис.3. Комплексная система реального времени в общем виде

По большому счету, проблема реального времени и понятие реального времени возникают именно в ходе функционирования комплексной СРВ, так как очень важны взаимодействия компонентов комплексной СРВ, а не только особенности работы одного из ее компонентов – вычислительной СРВ.

В модели большой комплексной системы человек-оператор может отсутствовать, и тогда мы имеем дело с комплексной СРВ, состоящей только из *двух* компонентов: системы управления и объекта управления (см. рис. 4).

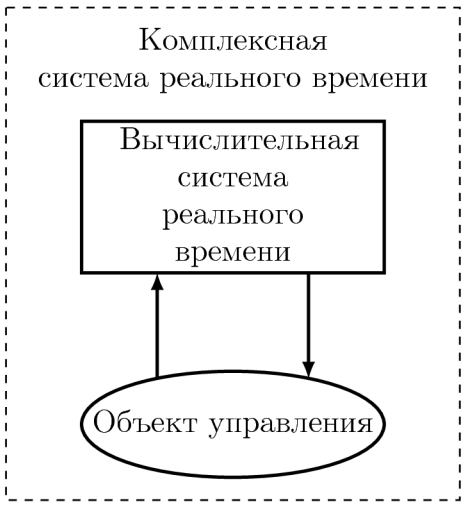

Рис. 4. Комплексная система реального времени, состоящая из двух компонентов

При этом следует выделить важный класс комплексных СРВ, в которых отсутствует человек-оператор, и вычислительная СРВ взаимодействует с другими комплексными системами реального времени (см. рис. 5).

Кроме того, объект управления также может взаимодействовать с другими комплексными СРВ (см. пунктирные стрелки на рис. 5), например, за счет физической связи с объектами управления в составе других комплексных СРВ.

Таким образом, комплексная СРВ может взаимодействовать с другими комплексными СРВ за счет взаимодействия как вычислительной СРВ, так и объекта управления.

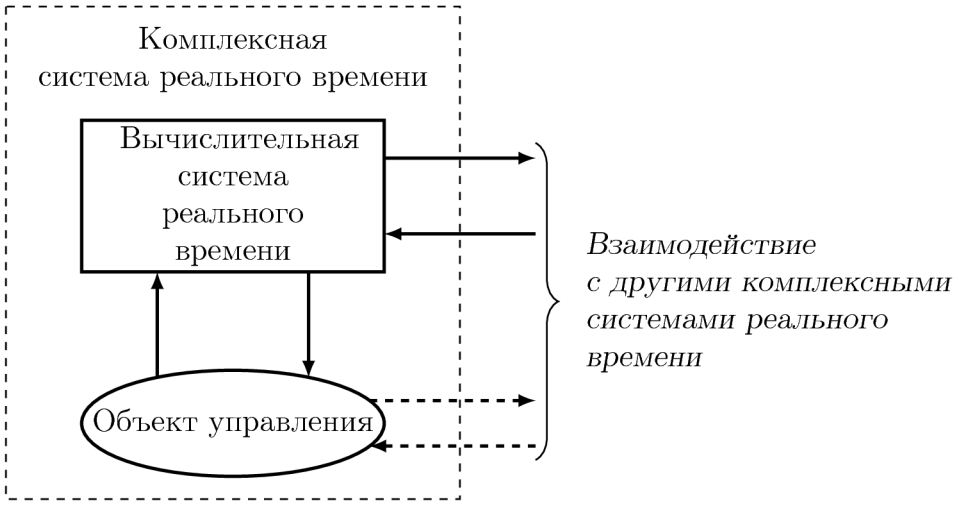

Рис. 5. Комплексная система реального времени, взаимодействующая с другими комплексными системами реального времени

Под термином СРВ (без предшествующего прилагательного) обычно понимают именно вычислительную СРВ. Однако в некоторых случаях термином СРВ также обозначают комплексную систему СРВ, например (см. рис. 1.2) в монографии [28].

Для краткости, в ходе дальнейшего изложения термином СРВ будет обозначать именно вычислительную СРВ.

1.1.6. Реальное время и быстродействие

Одно из распространенных заблуждений относительно СРВ состоит в том, что работа в реальном времени не всегда означает высокое быстродействие работы системы, и СРВ не обязательно должна быть быстродействующей. Может оказаться, что гораздо важнее, чтобы СРВ была предсказуемой в своем поведении и гарантировала соблюдение заданных ограничений РВ.

Например, пусть задано ограничение РВ, состоящее в том, чтобы команда на открытие клапана была сформирована не позднее чем, через 5 мс после поступления аварийного сигнала. И это ограничение является жестким, то есть никогда не должно нарушаться, иначе это может привести к поломке технологического оборудования. Пусть была реализована система, которая формирует команду с задержкой от 3 до 4 мс. Такая система действительно работает в реальном времени, так как ее функционирование гарантирует соблюдение заданного ограничения РВ, что и является оценкой соответствия реальной скорости протекания процессов во внешней среде. Пусть также была спроектирована вторая более быстродействующая система, которая формирует команду в ответ на аварийный сигнал с задержкой в 1 мс в подавляющем большинстве случаев, но иногда (пусть даже очень редко) эта задержка может стать равной 6 мс, например, из-за выполнения

некоторой вспомогательной программы. Для этой (второй) системы не гарантируется соблюдение заданного ограничения РВ, хотя в большинстве случаев она выдает команду быстрее, чем первая система. И можно сказать, что первая система функционирует правильно. А вот функционирование второй системы правильным нельзя назвать, хотя она в среднем является более быстродействующей, чем первая.

Хороший пример приведен в работе [69], и он описывает работу системы подушки безопасности в автомобиле. В момент удара подушка безопасности должна раскрыться не позднее определенного времени, чтобы успеть защитить человека. При этом она не должна раскрыться раньше определенного времени, так как если подушка раскроется слишком рано, то к моменту соприкосновения с человеком она может уже немного сдуться, что ослабит ее защитное действие. Поэтому в данном случае излишнее быстродействие может привести к неправильному функционированию СРВ. Также этот пример показывает, что ограничения РВ могут быть представлены не только в виде максимально допустимых значений (верхних границ) для интервалов времени, но и виде минимально допустимых значений (нижних границ).

1.1.7. Процесс разработки системы реального времени

Как и в случае разработки большинства систем, процесс разработки СРВ включает в себя два основных этапа: составление *спецификации* и получение *реализации*.

Существуют сложности в четком разграничении таких понятий, как спецификация и реализация СРВ. Традиционно спецификация указывает на то, *что* СРВ должна выполнять, а реализация – это то, *как* СРВ должна это выполнять. Но, с другой стороны, существует ряд формальных языков, каждый из которых способен описывать широкий спектр программ: от наиболее абстрактных (спецификаций) до исполняемых программ (реализаций). В

частности, поэтому трудно бывает отделить процесс спецификации от процесса реализации СРВ.

Также существует расширенное понимание спецификации, частично распространяемое и на реализацию. Формальная спецификация использует математическую нотацию, чтобы описать точным образом свойства, которые система должна иметь. Формальная спецификация *требований* – это описание того, что требуется от системы без детализации особенностей реализации. Формальная спецификация *реализации* – это описание того, как требуемые свойства системы должны быть достигнуты, каким образом они должны быть реализованы. Формальная *верификация* – это процесс показывающий, что формальная спецификация реализации удовлетворяет формальной спецификации требований.

Будем придерживаться следующих наиболее общих определений. *Спецификация* – это описание желаемого внешнего поведения системы. *Реализация* – это способ воплощения заданной спецификации в разрабатываемой системе. Под терминами «спецификация» и «реализация» будут пониматься именно *результаты* выполнения определенных этапов процесса разработки СРВ, а не *процессы* получения этих результатов.

Обычно процесс разработки СРВ выглядит следующим образом. Сначала составляется спецификация СРВ, включающая в себя в том числе перечень всех ограничений РВ. Эта спецификация либо может быть явным образом представлена в техническом задании на разработку СРВ, либо может быть составлена на основе технического задания. Первый вариант, конечно, представляется предпочтительным, так как наличие нечеткого описания требований к СРВ может привести к известным проблемам их различных толкований на последующих стадиях разработки. Основные особенности будущей реализации СРВ (архитектура, выбор аппаратно-программных решений) отражаются в дальнейших документах, формируемых в ходе процесса разработки СРВ. В полном объеме реализация СРВ должна быть от-

ражена в рабочей документации по окончании процесса разработки СРВ.

Важным моментом при разработке СРВ является проверка того, что реализация СРВ соответствует ее спецификации. Существуют два основных способа такой проверки:

– формальная верификация (формальное доказательство того, что формальное описание реализации удовлетворяет заданной спецификации);

– тестирование (всесторонние испытания функционирования реализованной СРВ при различных условиях).

Поскольку формальная верификация является трудоемким и громоздким процессом, то ее обычно применяют только по отношению к отдельным наиболее ответственным и критичным подсистемам, а для проверки остальных подсистем используют тестирование.

Обнаружение ошибок в реализации СРВ ведет к повторному анализу спецификации, частичной или полной переработке ранее полученной реализации.

Уменьшение затрат при разработке СРВ и, в частности, ускорение процесса разработки, уменьшение количества итераций при переходе от спецификации к реализации, упрощение формальной верификации и увеличение достоверности при тестировании – это основные проблемы, решаемые в рамках исследований, посвященных различным аспектам разработки СРВ.

1.1.8. Понятие управляющей системы реального времени

Управляющая СРВ – это СРВ, которая управляет некоторым объектом (объектом управления).

В системах, представленных на рис. 1, 3, 6, вычислительная СРВ является управляющей СРВ.

Таким образом, управляющая СРВ – это частный случай СРВ (частный случай вычислительной СРВ).

Как соотносятся между собой понятия управляющей СРВ и системы управления? Чтобы ответить на этот вопрос, достаточно вспомнить о том, что система управления техническим объектом может быть реализована не только на основе вычислительных устройств, а например, на основе аналоговых преобразователей. Отсюда сразу же приходим к выводу о том, что не всякая система управления является управляющей СРВ, при этом любая управляющая СРВ является системой управления.

Этот вывод справедлив только в том случае, если мы договариваемся о том, что СРВ – это обязательно *вычислительная* СРВ, то есть система основанная на вычислениях и реализованная с помощью тех или иных вычислительных устройств. Если же из определения СРВ мы уберем упоминание о вычислении, а оставим только требование своевременности (в дополнение к логической корректности), то тогда мы придем к выводу, что любая система управления – это управляющая СРВ, и наоборот. Таким образом, все зависит от исходного определения СРВ. По большому счету, это вопрос терминологии. Например, можно говорить о вычислительной СРВ как некотором частном случае СРВ, которая не обязательно должна быть основана на вычислениях. Или, в качестве противоположного примера исходной договоренности о терминах, можно считать, что СРВ обязательно должна быть основана на вычислениях, но при этом она является частным случаем некоторой «обобщенной» СРВ.

Но, как уже отмечалось, под СРВ мы понимаем вычислительную СРВ. Такой подход прослеживается в большинстве источников. В качестве примера можно привести базовые монографии по СРВ [28, 53].

Оставаясь в рамках данного подхода, логично считать, что управляющая СРВ – это система управления, реализованная на основе вычислительных устройств.

Важно напомнить, что управляющая СРВ (вычислительная СРВ) может взаимодействовать не только объектом управления, но и с человеком-

оператором (см. рис. 3), а также с другими управляющими СРВ (см. рис. 5).

В качестве примера рассмотрим автоматизированную систему управления технологическим процессом сборки сложного оборудования. В этом случае объектом управления является заводской цех вместе со всей совокупностью оборудования, сборочными роботами, заготовками и составными частям. Тогда в качестве системы управления выступает вычислительный комплекс промышленной автоматизации: компьютеры, промышленные контроллеры, устройства связи с объектом. Эту систему управления можно рассматривать как единое целое, а можно выполнить декомпозицию и изучать совокупность взаимодействующих систем управления. В первом случае мы будем иметь дело с одной большой управляющей СРВ, а во втором – со множеством управляющих СРВ, взаимодействующих между собой. И в том, и в другом случае, одна или несколько управляющих СРВ реализуют взаимодействие с человеком-оператором.

В качестве другого примера можно рассмотреть мобильного робота, который перемещается в незнакомой местности. Система управления этого робота, реализованная на основе микроконтроллеров, является управляющей СРВ, и она должна обеспечивать управление двигательными системами данного робота, с учетом цели перемещении и окружающей обстановки. При этом управляющая СРВ должна своевременно реагировать на события во внешнем мире, оказывающие влияние на функционирование робота. Кроме того, управляющая СРВ может обеспечивать удаленное взаимодействие с человеком-оператором, например, для получения указаний от него. В случае функционирования группы роботов важной особенностью управляющей СРВ является возможность взаимодействия с управляющими СРВ других роботов.

В дальнейшем вместо термина «управляющая СРВ» может употребляться термин «СРВ», что обусловлено, с одной стороны, стремлением к краткости, с другой стороны, тем, что многие утверждения, касающиеся

управляющих СРВ справедливы также и для всего класса СРВ.

1.2. Специфика организации программного обеспечения для управляющих систем реального времени

1.2.1. Задачи реального времени управляющих систем реального времени

1.2.1.1. Общие сведения

Управляющая СРВ должна выполнять определенные действия для внешних взаимодействий, например, с объектом управления, а также для изменения своего внутреннего состояния. Действия, которые должна выполнять система, задаются при составлении спецификации на основе совокупности выполняемых *задач*. При этом выполнение определенной задачи соответствует выполнению системой определенных действий. Такой подход реализует декомпозицию системы как совокупности взаимодействующих задач, что обеспечивает упрощение анализа, верификации и реализации системы.

Естественно, что уровень декомпозиции может выбираться согласно определенным критериям в процессе разработки СРВ. В частности, на одном этапе разработки может рассматриваться одна задача, реализующая определенную функцию СРВ, а на другом этапе эта задача может быть представлена как совокупность задач, каждая из которых реализует определенные алгоритмы, необходимые для осуществления требуемой функции СРВ.

В дальнейшем будут рассматриваться проблемы, обусловленные функционированием в РВ, поэтому, как правило, будут приниматься во внимание только задачи, связанные с РВ, обозначаемые для краткости как задачи РВ.

В реальном мире все происходит параллельно, и система, взаимодей-

ствующая с этим реальным миром, должна имитировать такое параллельное поведение. Поэтому следует обеспечивать параллельность или хотя бы имитацию параллельности выполнения задач.

Имитация параллельности достигается на основе разделения времени одного вычислительного устройства, при этом оно попеременно занято выполнением то одной, то другой задачи. Действительная параллельность (а не ее имитация) может обеспечиваться на основе совокупности параллельно работающих вычислительных устройств, каждое из которых предназначается для выполнения отдельной задачи. Однако реализация системы на основе множества параллельно работающих вычислительных устройств может оказаться существенно дороже, а также такая реализация часто сопряжена с существенными сложностями, вытекающими из необходимости обеспечения взаимодействия различных вычислительных устройств.

Многие особенности реализации, в частности, количество вычислительных устройств, часто не указываются в спецификации. Обычно в спецификации определяется необходимость параллельности выполнения задач без указания подробностей реализации этой параллельности.

Также в спецификации обычно не указывается основа реализации той или иной задачи. Выполнение задачи предполагает выполнение определенного алгоритма. В дальнейшем этот алгоритм может быть реализован: полностью аппаратно; полностью программно; частично аппаратно и частично программно. Программная реализация предполагает наличие одного или нескольких вычислительных устройств.

В дальнейшем основное внимание будет уделяться задачам на основе программной реализации в предположении, что используется только одно вычислительное устройство. Сужение области рассмотрения обусловлено тем, что, даже в случае многопроцессорных систем и систем со многими вычислительными устройствами зачастую остается проблема выполнения нескольких задач на отдельном устройстве в рамках такой комплексной сис-

темы.

Функционирование СРВ предполагает повторяемость, цикличность при выполнении задач.

Повторяемость и цикличность при выполнении задач обычно отражаются на основе представления задачи как последовательности *запросов*. В этом случае выполнение задачи (выполнение соответствующего алгоритма) заменяется выполнением последовательности запросов (выполнение отдельных алгоритмов через некоторые промежутки времени). В англоязычной литературе термину «запрос» соответствуют термины «request», «job», «task instance».

Как и в случае понятия задачи, трудно дать формальное определение понятию «запроса». Без потери общности и для удобства можно говорить о том, что задача формирует запросы, выполнение которых и реализует выполнение задачи. Например, если рассматривать задачу как некий периодически выполняемый алгоритм, то в этом случае выполнение очередного запроса соответствует очередному выполнению такого алгоритма.

Разделение процесса выполнения задачи на последовательность выполнения отдельных запросов обычно осуществляется на основе выделения совокупности повторяющихся действий, разделяемых интервалами времени, в течение которых задача не выполняется.

Также важно понять, в какой момент времени должен формироваться запрос, и по чьей инициативе он формируется. По отношению к этому вопросу различают следующие два основных типа задач:

– задачи, управляемые временем (time-driven). Запросы этих задач должны формироваться на основе анализа текущего времени в такие моменты времени, чтобы выполнялись ограничения РВ этих задач, указанные в спецификации;

– задачи, управляемые событиями (event-driven). Запросы этих задач должны формироваться в моменты возникновения событий, указанных в

спецификации.

Очень часто задачи, управляемые временем, называют *периодическими* задачами, так как среди задач, управляемых временем, пожалуй, наиболее распространенными являются задачи, запросы которых должны формироваться периодически, через определенные интервалы времени. Соответственно задачи, управляемые событиями, обычно называют *апериодическими* или *спорадическими* задачами. Более подробная информация об этих типах задач будет изложена в дальнейшем.

Хотя не все задачи, управляемые временем, формируют запросы строго периодически, но и в общем случае задачи, управляемые временем, могут именоваться периодическими задачами. Это можно обосновать тем, что запросы таких задач формируются в общем случае с периодами, которые динамически изменяются. В этом случае важно, что время появления запроса вычисляется, то есть можно говорить о том, что вычисляется текущий период формирования запроса.

Напротив, время появления апериодических и спорадических запросов определяется временем появления тех или иных событий.

В дальнейшем для удобства, а также следуя традиции, хотя и принимая во внимание известный компромисс при использовании этих понятий, задачи, управляемые временем, будут называться периодическими.

Задачи, управляемые событиями, будут называться апериодическими или спорадическими, при этом выбор термина «апериодический» и «спорадический» определяется дополнительными характеристиками задачи, управляемой событиями.

1.2.1.2. Задачи реального времени и их представление на разных уровнях системной организации

Задачи РВ могут быть реализованы различным образом, на различных уровнях системной организации, а именно, можно выделить два ос-

новных уровня, на которых могут быть представлены задачи РВ:

- на системном уровне (на уровне операционной системы);
- на прикладном уровне (на уровне приложения, выполняемого в данной операционной системе).

При реализации на системной уровне можно выделить два основных способа реализации задачи РВ:

- в виде *процесса* операционной системы;
- в виде *потока* отдельного процесса операционной системы.

Один из самых очевидных вариантов реализации – это использование в качестве задачи РВ отдельного процесса, выполняемого в рамках операционной системы. В этом случае можно использовать планировщик процессов, предоставляемый операционной системой. Недостатком данного способа является то, что надо каким-то образом обеспечивать обмен данными между различными процессами. На очень упрощенном уровне можно сказать, что каждый процесс – это отдельная программа со своими переменными. Поэтому находясь «внутри» одной программы (одного процесса) получить доступ к переменным другой программы (другого процесса) не так просто по сравнению с тем, как обеспечивается доступ к собственным переменным. Но, конечно, обмен между процессами можно вполне удобно обеспечивается соответствующими средствами.

Также задача РВ может быть реализована в виде потока (thread), который также в переводе иногда называют нитью. В рамках одного процесса может выполняться несколько потоков, и они могут рассматриваться как отдельные задачи РВ. Преимуществом использования потоков одного процесса является то, что каждый из потоков имеет доступ к адресному пространству данного процесса. То есть, если упрощенно, в частности, потоки могут обращаться к одной и той же переменной, определенной для процесса при написании программы, реализующей данный процесс.

Не вдаваясь в подробности, можно сказать, что в рамках операцион-

ных систем семейства linux различие между процессами и потоками «более плавное», чем в случае операционных систем семейства windows. То есть в случае windows имеется более резкое различие между реализацией задачи в виде процесса и реализацией задачи в виде потока. Поэтому при выборе способа реализации в виде потока или процесса надо также учитывать особенности операционной системы и того, как процессы и потоки представлены в рамках данной операционной системы.

Теперь обратимся к прикладному уровню.

Здесь надо отметить, что при реализации задачи РВ на прикладном уровне уже гораздо сложнее выполнить какую-то классификацию способов реализации задач.

С общих позиций можно сказать, что приложение (выполняемое как отдельный процесс) может выступать в роли своеобразного «заменителя», «имитатора» операционной системы. И в рамках данного приложения организуются своего рода «имитаторы» процессов, и уже для этих «процессов» реализуется имитация их параллельного выполнения подобно тому, как организуется параллельное выполнение настоящих процессов.

Например, в рамках приложения может быть несколько алгоритмов, оформленных в виде отдельных функций (в смысле языка программирования). Также в приложении имеется основной цикл, тело которого выполняется периодически с заданным интервалом времени, используя те или иные механизмы, например, таймер. В ходе выполнения этого тела цикла последовательно вызываются вышеуказанные отдельные функции. Тем самым, осуществляется циклическое выполнение функций. Обеспечивается повторение выполнения каждой функции. При необходимости вызов отдельных функций можно выполнять не на каждом цикле, а на каждом k-м цикле, задав изначально необходимое значение k. При таком подходе, каждая из указанных функций может рассматриваться как отдельная задача РВ, а очередное выполнение функции может считаться выполнением очередного

запроса соответствующей задачи.

Таким образом, задачи РВ могут быть представлены на различных уровнях организации программного обеспечения. Но на более высоких уровнях абстракции можно пренебрегать конкретными деталями представления задач РВ. Например, в рамках определенных моделей может быть не столь важно, как реализуется задача (на основе потока или процесса), и этим тогда пренебрегают. При этом обычно важным являются параметры задачи РВ, которые надо учитывать при работе с ними в рамках подобных моделей.

1.2.1.3. Запросы, формируемые задачами реального времени

Чтобы определить основные параметры запросов, необходимо рассмотреть особенности выполнения запроса в СРВ (см. рис. 7).

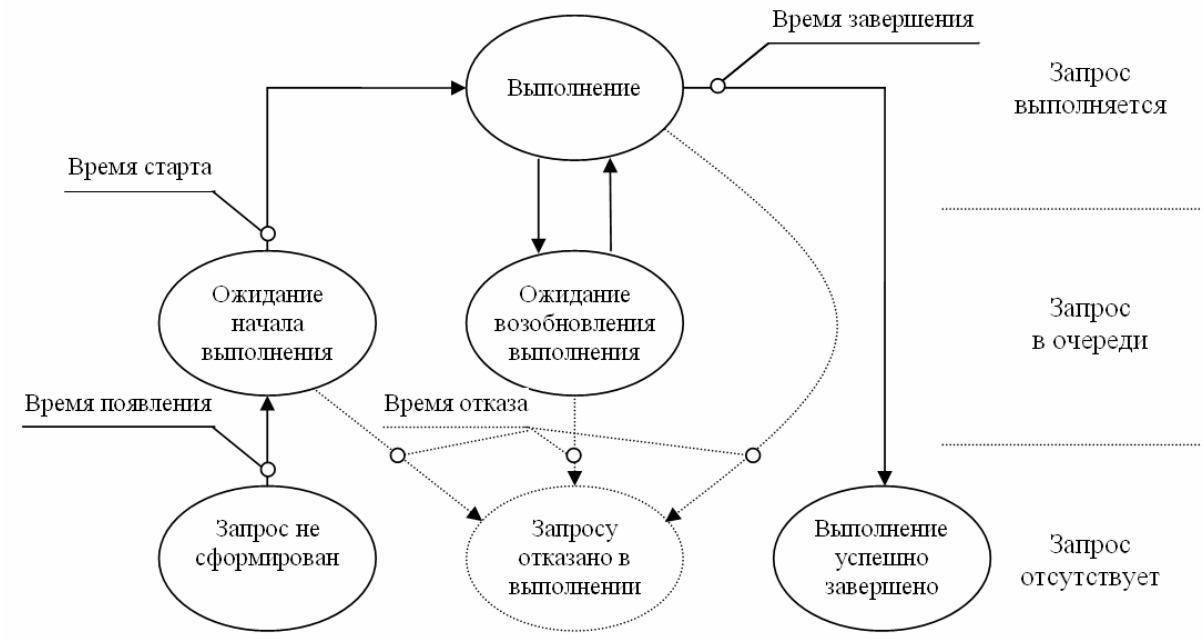

Рис.7. Граф состояний запроса, формируемого задачей реального времени

Первоначально запроса не существует. *Время появления* указывает на момент времени, когда запрос формируется задачей. После появления запрос находится в состоянии ожидания начала выполнения. *Время старта* отмечает начало выполнения запроса. Выполнение запроса может *преры-*

ваться, после прерывания запрос находится в состоянии ожидания возобновления выполнения. В общем случае цикл прерывания и возобновления выполнения запроса может осуществляться неоднократно. *Время завершения* - это момент времени, когда запрос заканчивает свое выполнение. После этого запрос переходит в состояние, когда его выполнение завершено.

Некоторым запросам может быть отказано в выполнении. При отказе происходит досрочное завершение выполнения запроса. При этом запрос удаляется, как и в случае успешного завершения. Естественно, что отказ может произойти только после старта запроса. *Время отказа* - это момент времени, когда происходит отказ в выполнении запроса.

Следует отметить, что каждый запрос РВ характеризуется *длительностью* его выполнения.

Время старта больше или равно времени появления и меньше времени завершения. Длительность запроса меньше или равна разнице между временем завершения и временем старта этого запроса. Необходимо учитывать, что выполнение запроса может прерываться.

Таким образом, основными параметрами запросов РВ являются:

- длительность;
- время появления;
- время старта;
- время завершения или время отказа.

На рис. 8 представлен пример диаграммы выполнения запросов РВ, а также указаны некоторые параметры выполнения запроса относительно временной оси.

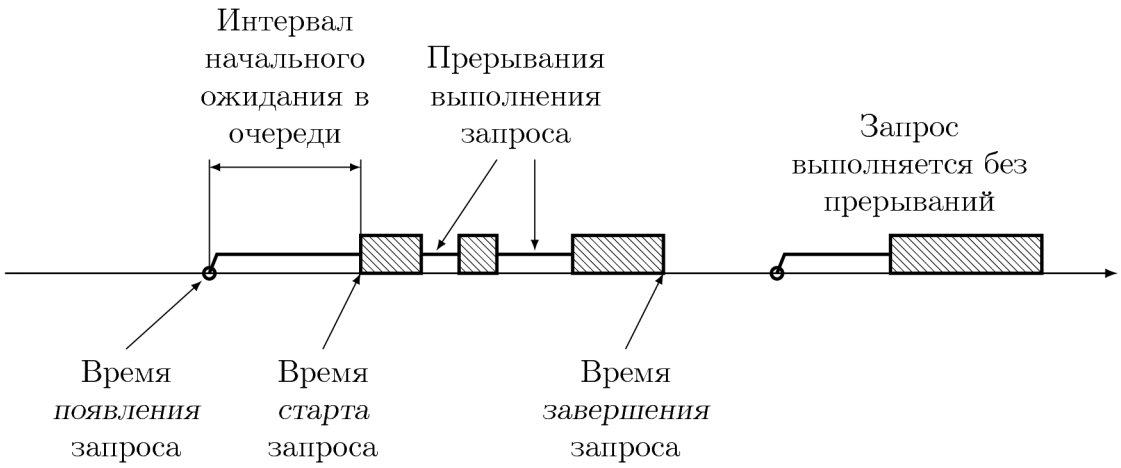

Рис. 8. Пример временной диаграммы выполнения запросов реального времени

Здесь и в дальнейшем при построении временных диаграмм используются следующие обозначения. Прямоугольниками обозначаются интервалы времени, когда происходит выполнения запроса. Окружность на временной оси обозначает время появления запроса. Линия на временной осью, соединяющая окружность и прямоугольник, обозначает интервал начального ожидания в очереди. Линия над временной осью, соединяющая прямоугольники, обозначает интервалы времени, когда происходит прерывание запроса.

В целом надо сказать, практически затруднительно, а в общем случае невозможно, указать параметры всех запросов в спецификации СРВ. Поэтому стремятся к тому, чтобы определить эти параметры через параметры задач.

1.2.1.4. Базовые параметры задач реального времени

Рассмотрим базовые параметры задач РВ. Здесь надо сказать, что в более сложных моделях эти параметры могут видоизменяться, а также могут добавляться дополнительные параметры. Но во многих базовых моделях используются именно те параметры, которые сейчас и рассмотрим подробнее.

Для периодической задачи, естественно, характерен *период*, который

определяет интервал времени между появлением запросов этой задачи. Кроме периода также часто указывается *начальное смещение* (начальная фаза), которое определяет время появления первого запроса данной периодической задачи.

Отличительная особенность апериодических и спорадических задач состоит в том, что они относится к задачам, управляемым событиями. Обычно заранее нельзя точно сказать, когда произойдет то или иное событие. Поэтому обычно заранее нельзя знать, когда сформируется запрос апериодической или спорадической задачи.

Чем же отличаются друг от друга спорадические и апериодические задачи? По большому счету они отличаются наличием информации о характере возникновения событий, оказывающих влияние на формирование их запросов. Если более кратко, то они отличаются информацией о характере формирования их запросов.

В случае спорадических задач обычно известен *минимальный период* формирования запросов, который определяет минимально возможный интервал между моментами появления соседних запросов данной задачи. Например, спорадическая задача может быть связана с событием поступления пакета данных по каналу связи. При этом можно заранее определить, учитывая объем пакета и скорость передачи по каналу, что между соседними (по времени) пакетами данных не может быть интервала времени меньше некоторого значения. Тогда это значение и будет минимальным периодом спорадической задачи.

В случае апериодических задач общего вида, обычно заранее нет информации о моментах появления их запросов. Поэтому для таких задач невозможно указать параметры, подобные периоду или минимальному периоду. Однако, это не значит, что о характере формирования запросов апериодической задачи вообще не может быть ничего известно. Например, может быть известна средняя загрузка процессора запросами этой задачи.

На основе значения этой загрузки можно косвенным образом получить некоторую информацию о характере формирования запросов. Главное здесь, что в случае апериодической задачи нельзя однозначно определить минимальный период, то есть минимально возможный интервал следования соседних запросов этой задачи. В частности, может оказаться, что этот минимальный период равен нулю.

Для всех видов задач (периодических, спорадических, апериодических) обычно заранее имеется некоторая информация о длительностях выполнения их запросов.

В идеальном случае заранее может быть известна точная длительность выполнения всех запросов задачи. Однако это бывает довольно редко. Обычно заранее известны некоторые оценки (верхние, нижние, средние) длительностей выполнения запросов. Чаще всего, заранее может быть известна максимальная длительность запросов, формируемых этой задачей. Тогда длительность любого запроса, формируемого этой задачей будет не больше этой максимальной длительности. В англоязычной литературе максимальной длительности выполнения запросов соответствует обозначение в виде аббревиатуры WCET, которая расшифровывается как worst-case execution time (время выполнения в наихудшем случае).

На рис. 9 представлены примеры выполнения задач РВ разных видов с указанием основных параметров этих задач.

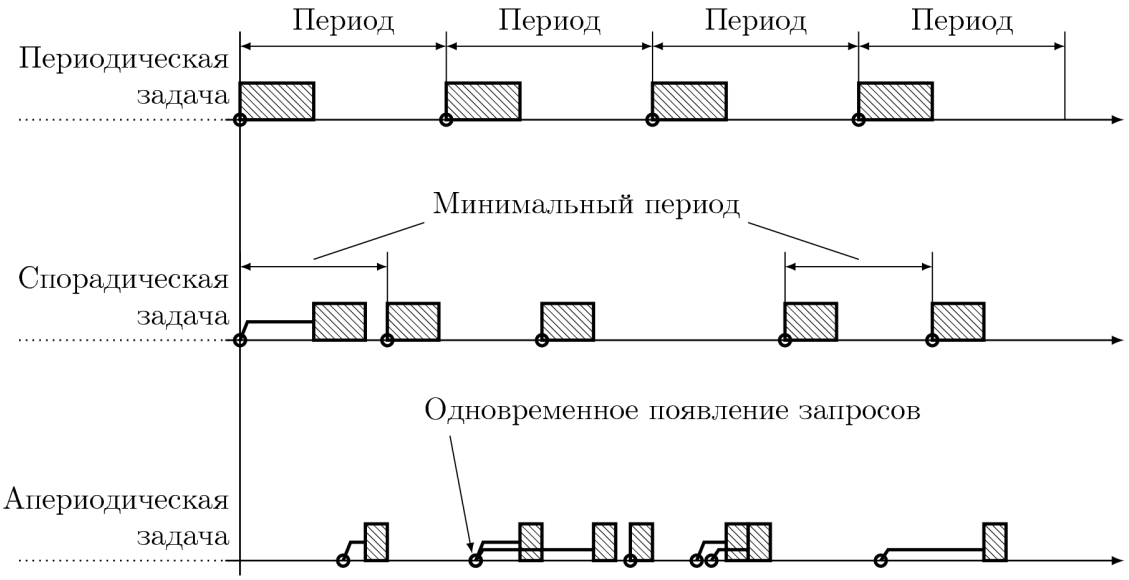

Рис.9. Пример выполнения задач реального времени разных видов

Видно, что в случае периодической задачи между моментами формирования запросов существует постоянный интервал, который и равен периоду.

В случае спорадической задачи интервалы между моментами появления запросов в общем случае различны. Но при этом существует некоторый минимально возможный интервал, который и равен минимальному периоду этой задачи.

В случае апериодической задачи заранее нет никаких сведений о величине интервалов между соседними запросами. В частности, может оказаться так, что в один и тот же момент времени могут появляться несколько запросов этой задачи, то есть может происходить одновременное появление нескольких запросов этой задачи.

На рисунке запросы каждой задачи имеют одинаковые длительности, но в общем случае, длительности отдельных запросов могут варьироваться.

Кроме того, на данной временной диаграмме смоделирована ситуация выполнения трех разных задач на одном процессоре. Нетрудно убедиться, что в каждый момент времени выполняется только какая-нибудь

одна задача (выполняется запрос данной задачи) или не выполняется ни одна из трех задач.

В такой ситуации время появления запроса может не совпадать с временем старта этого запроса из-за того, что в момент появления данного запроса процессор занят выполнением другого запроса.

1.2.1.5. Задачи и ограничения реального времени

Своевременность функционирования СРВ определяется своевременностью выполнения отдельных задач в составе СРВ. Поэтому требования РВ для СРВ предполагают формулирование требований РВ для каждой из задач в составе СРВ. Эти требования РВ удобнее всего определять в виде некоторых условий, так называемых ограничений РВ.

Можно даже говорить о том, что основное отличие СРВ от других систем, определяемых на основе совокупностей задач, - это наличие ограничений РВ для выполняемых задач.

С каждой задачей РВ связывается ограничение РВ. Например, может быть задано ограничение на максимально допустимую длительность выполнения каждого запроса задачи и на максимально допустимый период формирования запросов.

Ограничение РВ для задачи контура управления порождается на основе требований к качеству управления объектом. Ограничения формируются так, чтобы их соблюдение гарантировало приемлемый уровень качества управления.

В целом, получается, что ограничение РВ - это своего рода «посредник», «промежуточный язык», позволяющий перевести внешние требования в требования к задачам РВ.

Важно понимать, что при таком посредничестве обычно теряется часть информации, например, из-за разных упрощений. Это подобно тому, как теряется часть информации при переходе от реального объекта к моде-

ли объекта.

1.2.1.6. Жесткое и мягкое реальное время

Если нарушение ограничения абсолютно неприемлемо, то оно называется ограничением *жесткого* реального времени (ЖРВ).

Если нарушения ограничения допускаются, но нежелательны, то, скорее всего, мы имеем дело с ограничением *мягкого* реального времени (МРВ).

Удобно говорить об ограничении МРВ, как о некоторой функции качества, значение которой зависит от тех или иных характеристик времени выполнения запросов данной задачи.

Например, можно говорить о задаче МРВ, для которой надо минимизировать среднее время задержки при выполнении ее запросов. Другим вариантом может быть необходимость минимизации разброса значений задержки при выполнении ее запросов.

Обычно при работе с задачами МРВ применяется тактика «максимального усилия» (best effort), состоящая в том, что в ходе работы системы делается все возможное, чтобы улучшить значение функции качества. При этом, в общем случае, не дается никаких гарантий по достижению определенного значения функции качества.

Таким образом, можно выделить два класса ограничений РВ:

- ограничения ЖРВ;
- ограничения МРВ.

Когда говорят о жестком или мягком реальном времени, скорее всего, имеют в виду именно наличие ограничений жесткого или мягкого РВ. Например, говоря о задаче ЖРВ (hard real-time task) подразумевают задачу РВ, имеющую ограничение ЖРВ.

Или например, говоря о системе ЖРВ подразумевают СРВ, содержащую хотя бы одно ограничение ЖРВ.

1.2.1.7. Классификация и примеры задач реального времени

Применительно к задачам МРВ появляется дополнительное свойство их ограничения, а именно обязательность выполнения запроса даже после нарушения этого ограничения. Ведь в случае МРВ допустимы нарушения ограничений РВ. Поэтому возникает вопрос, если запрос еще не начался, а ограничение уже нарушено, то надо ли начинать выполнять запрос. По этому принципу разделяют задачи на обязательные (для которых запросы надо всегда выполнять) и необязательные (когда запросы не надо выполнять, если ограничение МРВ нарушено).

Следует принять во внимание особенности реализации необязательных задач МРВ. Как правило, необязательная задача МРВ рассматривается как составная часть процесса ЖРВ. В частности известен подход, основанный на модели процесса ЖРВ, состоящего в общем случае из двух задач ЖРВ и одной необязательной задачи МРВ [26].

Для получения информации о планировании необязательных задач МРВ полезно обратиться к работе [26].

Кроме различия задач МРВ по необходимости выполнения ограничений РВ (обязательные и необязательные задачи) существует их различие по способу формирования запросов, то есть выделяют периодические и апериодические задачи.

Таким образом, получаем различные виды задач РВ (таблица 1).

Таблица 1.

Классификация и примеры задач РВ

Вид задач РВ	Примеры задачи РВ данного вида
Периодические задачи ЖРВ	Периодическое измерения параметров и формирования периодических управляющих воздействий в контуре дискретного управления; контроль за поведением периодически измеряемого параметра; периодический обмен инфор-

	мацией по каналам связи; периодическое отслеживание аварийных ситуаций; периодический контроль за изменением состояния дискретных сигналов.
Спорадические задачи ЖРВ	Реакция на запрос по каналу связи; реакция на аварийные сигналы; реакция на изменение состояния контролируемого объекта.
Обязательные периодические задачи МРВ	Периодическое обновление отображаемой информации для интерфейса с оператором; периодическое измерение визуально контролируемых параметров; периодическая самодиагностика системы; периодическое архивирование параметров контролируемого объекта; обработка мультимедийных потоков информации.
Обязательные апериодические задачи МРВ	Реакция на команды оператора; реакция на предупредительные сигналы; реакция на удаленные запросы оператора по каналам связи.
Необязательные периодические задачи МРВ	Дополнительная часть в периодических процессах ЖРВ, например, реализующая более точные вычисления в концепции неточных вычислений [15,58];
Необязательные апериодические задачи МРВ	Дополнительная часть в спорадических процессах ЖРВ, например, реализующая более точные вычисления в концепции неточных вычислений [15,58];

1.2.2. Планирование задач реального времени

В ходе разработки программного обеспечения СРВ ограничения РВ учитываются при реализации *подсистемы планирования*. В общем случае *планирование* – это распределение ресурсов (памяти, времени доступа к процессору, к устройствам ввода-вывода) между запросами различных задач, направленное на соблюдение ограничений РВ. Распределение ресур-

сов осуществляется на основе определенного *алгоритма планирования*. В дальнейшем, если не будет особо оговариваться, то под планированием будет пониматься только распределение ресурса процессорного времени, т. е. распределение вычислительных ресурсов. Подсистема планирования для СРВ используется в составе применяемой операционной системы РВ или специально разрабатывается с учетом особенностей конкретной системы.

Многие исследования посвящены проблемам планирования задач РВ. Хороший обзор истории и современного состояния этих исследований приведен в работе [77]. Основные классические результаты при исследовании проблем планирования в СРВ рассматриваются в работе [84].

Планирование осуществляется на основе определенного алгоритма. *Исходными данными* для этого алгоритма являются данные о задачах РВ и об ограничениях РВ. *Результатом* выполнения этого алгоритма является распределение вычислительных ресурсов между запросами задач РВ. Но определенная часть информации известна уже до начала функционирования СРВ. Тогда часть необходимых вычислений может быть осуществлена до запуска СРВ.

Поэтому различают стадии *планирования* и *диспетчеризации*. Соответственно в составе алгоритма планирования можно выделить алгоритм планирования и алгоритм диспетчеризации.

Алгоритм планирования направлен на получение необходимых *промежуточных данных*, которые будут потом использоваться при диспетчеризации. К этому алгоритму не предъявляется особых требований быстродействия, так как он выполняется до начала функционирования СРВ.

Алгоритм диспетчеризации основывается на промежуточных данных, а также на информации, поступающей в ходе функционирования СРВ. В частности, необходимая информация об апериодических запросах, как правило, становится известной только в момент их появления. Алго-

ритм диспетчеризации должен занимать значительно меньше процессорного времени по сравнению с задачами РВ, иначе эффективность планирования может сильно снизиться.

Планирование задач ЖРВ и планирование апериодических запросов существенно различаются по своей сути. Действительно, в случае задач ЖРВ надо гарантировать соблюдение всех ограничений ЖРВ, а в случае апериодических запросов обычно устанавливается требование минимизации или максимизации некоторого интегрального параметра, характеризующего качество планирования апериодических запросов.

В составе алгоритма планирования задач ЖРВ можно выделить так называемый *анализ выполнимости* (feasibility analysis) [77] или его еще называют анализом планируемости или тестом планируемости (schedulability analysis или schedulability test) [28, 53]. Анализ выполнимости – это отыскание ответа на вопрос о том, гарантируется ли соблюдение ограничений ЖРВ при диспетчеризации с учетом полученных промежуточных данных. Если анализ выполнимости дает ответ, что выполнение ограничений ЖРВ не гарантируется, то возможны повторные попытки планирования, а также может потребоваться модификация аппаратного и программного обеспечения.

Для соблюдения ограничений ЖРВ прямо или косвенно выделяется определенный процент процессорного времени. Остальное время – это *свободное время*, используемое для выполнения апериодических запросов [37, 77].

Сложные СРВ содержат разные виды задач РВ. Одна из главных проблем планирования в сложных СРВ – это обеспечение баланса между *гибкостью* и *предсказуемостью* планирования [83].

Гибкость означает способность подсистемы планирования обрабатывать задачи с изначально недетерминированными параметрами, в частности обрабатывать апериодические запросы. Тогда гибкость тем больше,

чем больше эффективность планирования апериодических запросов. Предсказуемость определяется объемом предварительной информации об особенностях выполнения задач РВ. Предсказуемость тем больше, чем больше этой информации известно перед началом функционирования СРВ. При планировании задач ЖРВ минимально необходимым уровнем предсказуемости является информация о том, что ограничения ЖРВ не будут нарушены в ходе функционирования ЖРВ.

Существует множество вариантов разделения функций между стадиями планирования. От этого разделения зависят предсказуемость и гибкость планирования, а также сложность алгоритмов, объем промежуточных данных, эффективность планирования по различным критериям. С учетом основных особенностей такого разделения можно выделить *базовые модели планирования* (или они еще могут неформально называться *основными концепциями планирования*).

Подробнее о базовых моделях планирования см. п. 1.3.4.

1.2.3. Взаимодействие задач реального времени

В большинстве случаев задачи не являются в полном смысле независимыми. По большому счету задачи перестают быть независимыми, когда разделяют время одного вычислительного устройства, так как возможность выполнения одной задачи зависит от освобождения вычислительного устройства другой задачей. То же самое относится и к другим разделяемым ресурсам: памяти, каналам связи, устройствам ввода-вывода и т.д. Но задачи, которые являются «зависимыми» только от разделяемого вычислительного устройства, например, микропроцессора, по традиции называются независимыми. Это обусловлено тем, что в спецификации особенности использования вычислительных устройств, как правило, не указываются, так как эти особенности относятся к реализации. Поэтому в спецификации такие задачи являются независимыми.

Следует отметить, что зависимость задач может определяться спецификацией, а также может быть обусловлена особенностями реализации. Например, в спецификации могут явно указываться:

– разделяемые ресурсы, внешние по отношению к СРВ, взаимодействие с которыми СРВ должна обеспечивать на основе выполнения различных задач;

– причинно-следственные связи при выполнении задач такие, как порядок выполнения, условия выполнения одной задачи в зависимости от результатов выполнения другой задачи;

– передача между задачами информации, являющейся результатом выполнения задач.

Так или иначе, подобные зависимости задач определяют необходимость взаимодействия задач. При разработке СРВ должно обеспечиваться взаимодействие задач для выполнения требований, указанных в спецификации.

В ходе реализации может, например, возникнуть необходимость в некоторых общих для разных задач ресурсов: памяти, каналов связи, устройств ввода-вывода и т.д. Как и в случае спецификации, это также порождает зависимость задач и необходимость решения проблем, возникающих при взаимодействии этих задач.

1.3. Планирование задач реального времени

1.3.1. Проблема планирования задач реального времени

Система жесткого реального времени должна выполнять множество задач реального времени, соперничающих за ресурсы, так, чтобы все критичные (с точки зрения времени) задачи укладывались в отведенные им крайние сроки. Для выполнения каждой задачи требуются вычислительные ресурсы, ресурсы данных и прочие ресурсы, например, устройства ввода-

вывода. В монографии [54] приводится следующее определение проблемы планирования: проблема планирования – это проблема такого распределения указанных ресурсов, которое обеспечивает соблюдение всех требований реального времени.

Немного более детализированное определение используется в работе [28]. С небольшой переформулировкой и уточнением его можно воспроизвести здесь следующим образом.

Имеется множество из n задач $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, обозначаемое для также как $\{\tau_i\}$, множество из m процессоров $P = \{P_1, P_2, ..., P_m\}$, а также множество из q видов разделяемых ресурсов $R = \{R_1, R_2, ..., R_q\}$. Ограничения предшествования между задачами могут быть заданы в виде направленного ациклического графа. С каждой задачей могут связаны ограничения реального времени. Тогда *проблема планирования* – это проблема обеспечения такого назначения (за счет распределения по временной шкале) процессоров из P и ресурсов из R всем задачам из Γ, которое обеспечивает завершение (или выполнение) всех задач согласно заданным ограничениям.

Рассмотрим следующий пример, который иллюстрирует проблему планирования.

Пусть на однопроцессорном контроллере в составе системы управления выполняются две задачи РВ, обозначаемые τ_1, τ_2, для двух независимых контуров управления (рис 10а). Каждая i-я задача должна периодически формировать запросы $(\tau_{i,1}, \tau_{i,2}, ...)$, и запросу требуется время выполнения (здесь для простоты оно считается постоянным) для формирования очередного воздействия на объект (рис. 10б). В случае отдельного процессора для каждой задачи проблем не возникает, и они выполняются, как показано на рис. 10б. Однако при общем процессоре возникает взаимовлияние. Предполагается, что для разделения процессорного времени между

задачами применяется концепция планирования с фиксированными приоритетами (ПФП). Информацию о теоретических основах, методах, сферах практического применения ПФП можно найти в работах [27, 71, 77, 93] Пусть начальные смещения (O_1, O_2) и периоды (T_1, T_2) менять нельзя, тогда в рамках ПФП остается лишь менять приоритеты (π_1, π_2). Поэтому существуют только два варианта планирования: (π_1 выше π_2), (π_1 ниже π_2), приводящие к двум вариантам выполнения задач (см. рис. 10в). Согласно ПФП запрос задачи с более высоким приоритетом прерывает выполнение запроса задачи с более низким приоритетом. На рисунке окружностью отмечается момент ($r_{i,j}$) появления запроса $\tau_{i,j}$ в очереди запросов, а прерывание запроса обозначается линией над соответствующей временной осью. При этом 1-й вариант приводит к значительному нарушению строгой периодичности для τ_2 ($r_{2,3} \neq s_{2,3}$), где $s_{2,3}$ – начало выполнения $\tau_{2,3}$). Простое изменение приоритетов, приводящее ко 2-му варианту, обеспечивает строгую периодичность для τ_2, и небольшое нарушение периодичности для τ_1. Пусть известно, что такое небольшое нарушение для τ_1 оказывается допустимым. Тогда решением проблемы планирования будет 2-й вариант с соответствующими значениями (O_1, T_1, π_1), (O_2, T_2, π_2).

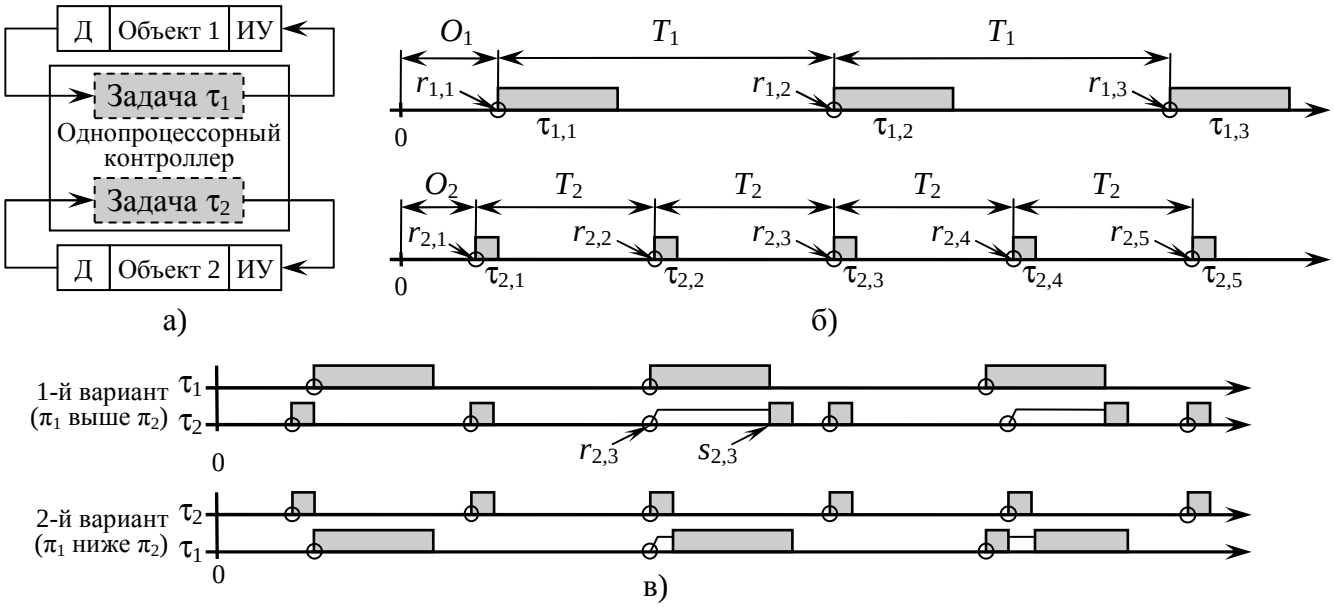

Рис. 10. Пример планирования задач РВ (Д – датчик; ИУ – исполнительное устройство)

Однако в приведенном примере имеется всего два варианта планирования, и из них легко выбрать подходящий. Возможность изменения O_i, T_i в заданных диапазонах, а также наличие $n!$ вариантов назначения приоритетов для n задач приводят к тому, что полный перебор вариантов становится практически нереализуемым. Известно, что указанная проблема планирования является NP-трудной [1]. Поэтому естественный подход к ее решению – это разработка эффективных эвристических алгоритмов, которые за приемлемое время с высокой вероятностью находят значения $\{O_i, T_i, \pi_i \mid i \in [1, n]\}$, обеспечивающие *подходящий* вариант планирования. Вариант планирования считается подходящим, если при его использовании для каждой задачи τ_i выполняется ее исходное ограничение РВ. Очевидно, что сложность разработки таких алгоритмов зависит от видов исходных ограничений РВ, которые имеют задачи.

1.3.2. Эффективность реализации системы реального времени и планирование задач реального времени

Разработанная СРВ должна полностью удовлетворять заданной спецификации. Реализация требований, указанных в спецификации, может быть выполнена с той или иной степенью *эффективности*.

Эффективность реализации определяется затратами:

– на приобретение готовых аппаратно-программных средств;

– на разработку СРВ, включающую разработку всех видов обеспечения СРВ, а также на монтажные и пуско-наладочные работы.

Если требования РВ не удается выполнить в заданных условиях, то имеются три варианта:

– выбор более быстродействующих вычислительных ресурсов;

– оптимизация программного обеспечения по быстродействию;

– более эффективное использование вычислительных ресурсов на основе более эффективного планирования задач РВ.

Первый вариант предполагает увеличение затрат на приобретение готовых аппаратных и программных средств. Использование второго варианта существенно увеличивает затраты на разработку программного обеспечения. Все это приводит к уменьшению эффективности реализации СРВ.

Алгоритмы более эффективного планирования задач РВ либо могут входить в состав готового программного обеспечения, либо должны быть реализованы в ходе разработки программного обеспечения. Поэтому дополнительные трудозатраты при третьем варианте – это либо настройка необходимых параметров в составе подсистемы планирования, либо реализация уже известных алгоритмов в составе разрабатываемого программного обеспечения. Во многих случаях именно этот вариант является наи-

менее затратным, что позволяет его рассматривать в качестве первоочередного.

Так или иначе, эффективность реализации СРВ во многом зависит от эффективности планирования задач РВ. Применение более эффективного планирования позволяет избежать дополнительных затрат на оптимизацию по быстродействию разрабатываемого программного обеспечения и на приобретение более быстродействующих вычислительных средств.

Таким образом, разработка алгоритмов эффективного планирования – это важнейшая составляющая проблемы повышения эффективности реализации управляющей СРВ.

1.3.3. Критерии планирования

Степень эффективности планирования можно определить только на основе некоторых *критериев* планирования.

Результатом планирования является расписание, которое формируется в явном или неявном виде. Расписание определяет распределение ресурсов между задачами.

Проблемы оптимального планирования или составления оптимальных расписаний рассматриваются в теории расписаний [9, 10]. Очевидно, что в основе этих проблем находятся критерии планирования или критерии составления расписания.

В теории расписаний обычно используются такие критерии, как минимизация суммы времен завершения работ, минимизация взвешенной суммы времен завершения работ, минимизация длины расписания, минимизация количества требуемых процессоров, минимизация максимального запаздывания. В большинстве случаев ограничения РВ не рассматриваются.

Естественно, что критерии, на основе которых составляется расписание, существенным образом зависят от практических целей и задач. На-

пример, необходимость минимизации длины расписания относится к статическим системам, не учитывающим РВ, а минимизация задержек и увеличение производительности – это главные критерии в динамических системах, также не учитывающих РВ. Однако в СРВ основной целью является соблюдение ограничений РВ, поэтому и критерии планирования существенно отличаются в случае СРВ [45].

Критерии планирования в СРВ существенным образом зависят типа задач РВ, то есть от степени необходимости выполнения ограничений РВ.

Для задач ЖРВ необходимо гарантировать соблюдение всех ограничений РВ. Чем меньше требовательность в быстродействии вычислительных ресурсов для такого гарантирования, тем больше эффективность планирования. Планирование наиболее эффективно в том случае, если используется так называемый *оптимальный алгоритм планирования задач ЖРВ*. Это такой алгоритм, который не может обеспечить соблюдение всех ограничений ЖРВ только в том случае, когда никакой другой алгоритм планирования также не может этого обеспечить [93].

Для обязательных задач МРВ в общем случае необходимо обеспечивать достижение требуемого значения функции качества, зависящей от нарушения ограничений при выполнении запросов этих задач. Например, эта функция качества может зависеть от интервалов запаздывания при выполнении запросов. Чаще всего для упрощения анализа в качестве функции качества используют так называемое *среднее время задержки запросов*, которое определяется как среднее арифметическое интервалов запаздывания всех запросов обязательных задач МРВ на некотором достаточно большом интервале времени при условии, что относительный крайний срок для каждого запроса равен его длительности [35].

Для необязательных задач МРВ в общем случае необходимо обеспечивать достижение требуемого значения функции качества, зависящей от количества успешно выполненных запросов этих задач, ведь выполнение

некоторого количества этих запросов может быть завершено досрочно. Чаще всего для упрощения анализа в качестве функции качества используют *процент успешно выполненных запросов* среди всех сформированных запросов необязательной задачей МРВ на некотором достаточно большом интервале времени [35].

Вообще можно выделить следующие основные критерии.

Критерий ЖРВ – наличие оптимального алгоритма планирования задач ЖРВ.

Критерий обязательного МРВ – достижение минимального значения среднего времени задержки запросов, формируемых обязательными задачами МРВ.

Критерий необязательного МРВ – достижение максимального процента успешно выполненных запросов, формируемых необязательными задачами МРВ.

1.3.4. Базовые модели планирования задач реального времени

1.3.4.1. Планирование на основе фиксированного расписания

Планирование на основе фиксированного расписания также можно называть *табличным планированием* (table-driven scheduling). И в дальнейшем, для краткости (и чтобы не путать с термином «планирование с фиксированными приоритетами») будет использоваться термин «табличное планирование».

Табличное планирование предполагает составление расписания на этапе предварительного планирования. В этом расписании, обычно называемом таблицей, указывается распределение выполнения периодических запросов по времени. Алгоритм планирования обеспечивает составление расписания для задач ЖРВ. Анализ выполнимости определяется уже самим фактом наличия расписания, согласно которому все ограничения ЖРВ

соблюдаются. Промежуточные данные – это расписание. Диспетчеризация осуществляется на основе этого расписания. Периодические запросы выполняются согласно заранее составленному расписанию. Для выполнения апериодических запросов используется время, которое не было использовано в расписании, т. е. свободное время.

В общем случае проблема составления расписания в виде таблицы является NP-трудной [9, 10]. Поэтому алгоритм планирования может иметь значительную временную сложность при данной концепции планирования.

Табличное планирование обладает наибольшей предсказуемостью в первую очередь по отношению к периодическим задачам ЖРВ (обычно явным образом устанавливается время начала и окончания, а также прерывания выполнения каждого запроса), но при этом обеспечивает наименьшую гибкость из-за сложности в планировании апериодических запросов.

Преимущество табличного планирования – это высокая предсказуемость. Также преимуществом табличного планирования является возможность планирования при наличии сложных ограничений для задач РВ, а также при условии сложных взаимодействий между задачами, так как все это можно учесть в расписании, составляемом до начала функционирования СРВ.

Одним из недостатков табличного планирования является проблема длины расписания (таблицы), так как при наличии большого количества задач РВ с разными периодами длина таблицы часто оказывается неприемлемой для практических реализаций. Другой недостаток – это сложность выполнения апериодических запросов. Время появления каждого апериодического запроса заранее неизвестно, поэтому невозможно заранее составить таблицу, учитывающую выполнение этих запросов. Поэтому время задержки при выполнении апериодических запросов может быть значительным из-за необходимости ожидания интервала свободного времени.

Частично улучшить эту ситуацию помогли решения, основанные на динамическом изменении таблицы [45] .

Информацию о теоретических основах, методах, сферах практического применения табличного планирования можно найти в [8, 9, 10, 45, 54, 71].

1.3.4.2. Планирование с фиксированными приоритетами

Планирование с фиксированными приоритетами (ПФП) (fixed priority scheduling) предполагает формирование очереди, куда размещаются запросы. Из этой очереди запросы выбираются для выполнения согласно установленным для них приоритетам. Алгоритм планирования обеспечивает установку приоритета для каждой задачи ЖРВ, а также такие параметры, как начальное смещение и период. При этом приоритеты являются фиксированными, т. е. они не меняются в ходе функционирования СРВ. Промежуточные данные – это начальные смещения, периоды и приоритеты задач ЖРВ. Диспетчеризация осуществляется на основе формирования запросов периодических задач согласно установленным начальным смещениям и периодам, а также на основе обслуживания очереди сформированных запросов, учитывая фиксированные приоритеты периодических задач. Следует отметить, что приоритет запроса равен приоритету формирующей его задачи.

В случае ПФП анализ выполнимости может быть достаточно сложным, ведь в отличие от табличного планирования в данном случае отсутствует расписание. В ходе анализа выполнимости выясняется гарантированность соблюдения ограничений ЖРВ на основе установленных параметров задач ЖРВ.

Отсутствие расписания в случае ПФП уменьшает предсказуемость, но увеличивает гибкость по сравнению с табличным планированием, так как при этом подсистема планирования может более эффективно реагиро-

вать на появление апериодических запросов. Это объясняется тем, что в случае ПФП начало и окончание каждого запроса ЖРВ изначально не фиксированы, что обеспечивает возможность более простых способов выделения свободного времени для апериодических запросов. Информацию о теоретических основах, методах, сферах практического применения ПФП можно найти в [27, 71, 77, 93].

1.3.4.3. Планирование с динамическими приоритетами

Планирование с динамическими приоритетами (dynamic priority scheduling) аналогично ПФП за исключением того, что приоритеты периодических запросов меняются при диспетчеризации согласно тому или иному алгоритму динамического изменения приоритетов. Такой алгоритм выбирается в ходе планирования. Диспетчеризация осуществляется на основе формирования запросов периодических задач согласно установленным начальным смещениям и периодам, а также на основе обслуживания очереди сформированных запросов, учитывая динамическое изменение приоритетов периодических запросов согласно выбранному соответствующему алгоритму.

Аналогично ПФП в случае планирования с динамическими приоритетами большое значение уделяется анализу выполнимости. В ходе этого анализа выясняется гарантированность соблюдения ограничений РВ для задач РВ на основе выбранного алгоритма динамического изменения приоритетов.

Планирование с динамическими приоритетами обладает существенной гибкостью, обеспечивая большую, по сравнению ПФП, эффективность планирования апериодических запросов, но при этом имеет меньшую предсказуемость, чем ПФП. Информацию о теоретических основах, методах, сферах практического применения планирования с динамическими приоритетами можно найти в [12, 29, 71, 82].

1.3.5. Сравнение основных концепций планирования

Планирование должно обеспечивать такое распределение ресурсов времени между задачами, чтобы удовлетворить некоторые заранее установленные ограничения РВ. Одна из основных проблем, решаемых в рамках исследований СРВ, – это проблема соблюдения ограничений РВ. Поэтому планирование – это, возможно, наиболее интенсивно исследуемый раздел в области СРВ.

Следует отметить, что существенная часть проблем планирования в СРВ является NP-полной или NP-трудной.

Основные теоретические результаты исследований проблемы планирования в СРВ можно найти в обзоре [77].

При решении проблем планирования наилучшие результаты получены для СРВ, содержащих только периодические задачи. Это неудивительно, так как об этих задачах РВ имеется наибольшее количество информации на этапе проектирования, то есть планирование получается в большой степени детерминированным.

Но наряду с периодическими задачами в *сложных* СРВ присутствуют апериодические задачи. При этом в составе большинства сложных СРВ выделяют задачи с различными видами ограничений, а именно задачи ЖРВ, обязательные и необязательные задачи МРВ. Наличие задач с различными характеристиками и ограничениями приводит к проблеме эффективности *совместного планирования* этих задач. Указанная проблема становится все более актуальной из-за растущего в настоящее время количества разрабатываемых сложных СРВ.

Основные трудности при решении этой проблемы связаны с необходимостью *совмещения* разных критериев планирования (см. п. 1.3.3). Дело в том, что критерий ЖРВ требует высокой предсказуемости планирования, а критерии обязательного и необязательного МРВ требуют увеличения

гибкости планирования. Но в большинстве случаев предсказуемость и гибкость являются взаимно противоречащими свойствами планирования.

Как уже отмечалось, одна из основных проблем планирования – это обеспечение оптимального соотношения между гибкостью и предсказуемостью планирования. Основные концепции планирования реализуют различные варианты такого соотношения на основе различных вариантов разделения функций между алгоритмами в составе алгоритма планирования. Сравнительные характеристики основных концепций планирования представлены в таблице 2.

Таблица 2

Сравнительные характеристики базовых моделей планирования

Сравнительная характеристика	Основные концепции планирования		
	Табличное планирование	Планирование с фиксированными приоритетами (ПФП)	Планирование с динамическими приоритетами
Алгоритм планирования	Построение расписания (таблицы) выполнения периодических задач	Назначение периодическим задачам фиксированных приоритетов	Выбор алгоритма динамического изменения приоритетов
Анализ планируемости	Фактически отсутствует, так как определяется уже самим фактом наличия расписания	Выясняется возможность соблюдения ограничений РВ на основе установленных фиксированных приоритетов	Выясняется возможность соблюдения ограничений РВ на основе выбранного алгоритма динамического изменения приоритетов

Промежуточные данные	Расписание выполнения периодических задач	Фиксированные приоритеты периодических задач	Алгоритм динамического изменения приоритетов
Диспетчеризация	Выполнение запросов согласно составленному расписанию для периодических задач	Обслуживание очереди запросов, учитывая фиксированные приоритеты периодических задач	Обслуживание очереди запросов, учитывая алгоритм динамического изменения приоритетов
Предсказуемость	Наибольшая предсказуемость, так как в заранее составленном расписании содержится большой объем информации о выполнении запросов	Предсказуемость обеспечивается возможностью анализа порядка выполнения запросов, так как заранее известны фиксированные приоритеты	Наименьшая предсказуемость, так как сложно предсказать порядок выполнения запросов при динамическом изменении приоритетов этих запросов
Гибкость	Наименьшая гибкость, так как сложно выделить свободное время апериодических запросов в условиях расписания, которое уже составлено	Гибкость ограничивается фиксированными приоритетами, которые ограничивают свободное время для планирования апериодических запросов	Наибольшая гибкость, так как обеспечивается наиболее эффективное выделение свободного времени для апериодических запросов

Также, говоря о классификации моделей планирования, можно отметить, что табличное планирование относится к так называемым моделям оффлайнового планирования. А модели ПФП и планирования с динамиче-

скими приоритетами – к так называемым моделям онлайнового планирования.

Теперь важно рассмотреть более подробно, каким образом формально определяется базовая модель планирования. Сделаем это на именно модели ПФП, так как в настоящее время большее внимание уделяется ПФП [27,26], обеспечивающему компромисс между предсказуемостью и гибкостью планирования в СРВ (см. таблицу 1). Эта концепция планирования имеет хорошую теоретическую основу и множество методов, обеспечивающих его практическое применение в СРВ [27, 28, 71, 77, 93].

1.3.6. Базовая модель планирования с фиксированными приоритетами

1.3.6.1. Общие положения

В настоящее время все большее внимание уделяется концепции ПФП [77], обеспечивающей компромисс между предсказуемостью и гибкостью планирования в СРВ (см. п. 1.3.4.2). ПФП имеет хорошую теоретическую основу и множество методов, обеспечивающих его практическое применение в СРВ (см. например, [16, 18, 37, 39, 57, 76, 78, 91, 92, 93, 94, 99], а также соответствующий раздел обзора [77]). Кроме того, ПФП является очень удобной основой для дальнейшего уточнения модели планирования с целью повышения эффективности планирования. В качестве подобных примеров можно привести работы [22, 25, 26, 36, 46, 58, 87, 90].

В дальнейшем в качестве концепции планирования будет рассматриваться именно ПФП. При этом ниже описывается стандартная модель ПФП, которая с незначительными вариациями встречается в работах по проблемам ПФП.

Любой момент времени однозначно определяется некоторым действительным числом. При этом момент начала функционирования СРВ соот-

ветствует числу 0. *Наблюдаемый интервал функционирования СРВ* – это интервал времени в течение которого наблюдаются и оцениваются характеристики функционирующей СРВ, в том числе характеристики планирования, и этот интервал времени будет обозначаться $[0, t_{\max}]$, где $t_{\max}$ – *максимальный наблюдаемый момент времени*.

В дальнейшем для краткости каждый момент времени будет отождествляться с соответствующим действительным числом. Поэтому, например, для напоминания об этом может использоваться выражение вида «некоторый момент времени принадлежит $\mathbf{R}$», где $\mathbf{R}$ – множество всех действительных чисел. Также для краткости применительно к какому-нибудь событию может использоваться слово «время» вместо словосочетания «момент времени».

Шаг времени при планировании – это такое действительное число ε, что все события, определяемые планированием, могут возникать только в моменты времени $k\varepsilon \geq 0$, где k – целое число, и каждый из указанных моментов времени может быть моментом возникновения какого-нибудь события, определяемого планированием. В частности, получается, что наименьший интервал между двумя последовательными событиями равен ε. В дальнейшем для краткости будет использоваться выражение вида «значение кратно ε», которое считается эквивалентным выражению «значение равно $k\varepsilon$, где k – целое число».

Предполагается, что вычислительная система – однопроцессорная, и прерывания задач разрешены, при этом задачи являются независимыми.

1.3.6.2. Задачи жесткого реального времени

Имеется множество из n *задач ЖРВ* $\mathbf{\Gamma} = \{\tau_1, \ldots, \tau_n\}$, которое в дальнейшем будет обозначаться $\{\tau_i\}$. В дальнейшем предполагается, что всегда $n \geq 1$. Каждая задача τ_i имеет ограничение ЖРВ и она может быть либо

периодической, либо *спорадической*. Задача τ_i время от времени формирует *запросы*, и в общем случае τ_i формирует множество запросов, которые также могут называться запросами ЖРВ. Каждый j-й запрос, обозначаемый $\tau_{i,j}$, при своем появлении помещается в очередь запросов. При этом $r_{i,j}$ – это *время появления* запроса $\tau_{i,j}$. *Время старта* $s_{i,j}$ – это время начала выполнения запроса $\tau_{i,j}$, которое в общем случае определяется в ходе диспетчеризации согласно ПФП. *Время завершения* $f_{i,j}$ – это время завершения выполнения запроса $\tau_{i,j}$, которое также в общем случае определяется в ходе диспетчеризации согласно ПФП. Естественно, что $r_{i,j} \in \mathbf{R}$, $s_{i,j} \in \mathbf{R}$, $f_{i,j} \in \mathbf{R}$, и эти значения кратны ε. Множество всех запросов всех периодических и спорадических задач жесткого реального времени обозначается $\mathbf{\Omega(\Gamma)}$.

Выполнение каждого запроса может прерываться выполнением других запросов, но не может прерываться подсистемой планирования. Выполнение очередного запроса задачи τ_i не может начаться, пока не завершено выполнение всех предыдущих запросов этой задачи, тогда для $\forall j \geq 1$ справедливо соотношение

$$0 \leq s_{i,j} < f_{i,j} \leq s_{i,j+1} < f_{i,j+1} \ . \qquad (1.1)$$

Для упрощения дальнейшего изложения предполагается, что значения $r_{i,0}, s_{i,0}, f_{i,0}$ определены, и выполняется соотношение

$$r_{i,0} \leq s_{i,0} < f_{i,0} \leq 0 \ . \qquad (1.2)$$

Периодическая задача τ_i характеризуется тем, что при $\forall j \geq 1$ справедливо следующее соотношение

$$r_{i,j} = r_{i,j-1} + T_i = O_i + (j-1)T_i \ , \qquad (1.3)$$

где O_i – *начальное смещение* задачи τ_i; T_i– *период* формирования запросов периодической задачи τ_i. Предполагается, что O_i, T_i кратны ε. Кроме того, для удобства, здесь и в дальнейшем предполагается, что $r_{i,0} = O_i - T_i$, если задача τ_i является периодической.

Спорадическая задача τ_i характеризуется тем, что при $\forall j \geq 1$ справедливо следующее соотношение

$$r_{i,j} \geq r_{i,j-1} + T_i \ , \tag{1.4}$$

где T_i – *минимальный период* формирования запросов спорадической задачи τ_i. Очевидно, значение T_i всегда можно сделать кратным ε. Кроме того, для удобства, здесь и в дальнейшем предполагается, что если задача τ_i является спорадической, то $O_i = 0$, $r_{i,0} = -T_i$. Важно отметить, что если задача τ_i является спорадической, то значения $r_{i,j}$ при $\forall j \geq 1$ определяются внешними событиями и заранее неизвестны, в отличие от периодической задачи. При этом соотношение (1.4) указывает только на минимально возможный интервал между моментами появления любых соседних запросов спорадической задачи.

Пусть $C_{i,j}$ обозначает *длительность* $\tau_{i,j}$, т. е. сколько времени выполняется запрос $\tau_{i,j}$ при отсутствии прерывания его выполнения. Другими словами, $C_{i,j}$ – это значение $f_{i,j} - s_{i,j}$ при отсутствии прерывания $\tau_{i,j}$ Значение $C_{i,j}$ в общем случае становится известным после момента $f_{i,j}$. Естественно, что в любом случае $C_{i,j}$ строго больше нуля и является кратным ε, поэтому в любом случае $C_{i,j} \geq \varepsilon$.

Во многих случаях бывает сложно заранее определить $C_{i,j}$, так как количество шагов алгоритма может сложным образом зависеть от входных данных и некоторых внутренних переменных. Поэтому, чтобы заранее гарантировать соблюдение СО, вместо него обычно используется *верхняя*

оценка возможных значений $C_{i,j}$ при $\forall j$, т. е. значение C_i^{Up} такое, что $C_{i,j} \le C_i^{Up}$ при $\forall j$. Также во многих случаях может использоваться нижняя оценка возможных значений $C_{i,j}$ при $\forall j$, т. е. значение C_i^{Lo} такое, что $C_i^{Lo} \le C_{i,j}$ при $\forall j$. Таким образом, всегда выполняется соотношение $C_{i,j} \in [C_i^{Lo}, C_i^{Up}]$ при $\forall j$.

Очевидно, что значения C_i^{Lo}, C_i^{Up} всегда можно уточнить так, что они будут строго больше нуля и кратными ε. Поэтому в дальнейшем предполагается, что значения C_i^{Lo}, C_i^{Up} всегда строго больше нуля и являются кратными ε.

Часто значение C_i^{Up} обозначается просто C_i.

В дальнейшем предполагается, что на длительность запросов нельзя повлиять средствами подсистемы планирования.

В условиях ПФП каждая задача τ_i имеет определенный приоритет.

Пусть π_i – это целое число в интервале $[1, n]$, характеризующее приоритет задачи τ_i, и чем меньше значение π_i, тем выше приоритет τ_i. Также предполагается, что все приоритеты являются различными. Получается, каждая задача имеет приоритет в виде уникального числа из интервала $[1, n]$. Таким образом, множеству $\{\tau_i\}$ ставится в соответствие множество $\{\pi_i\}$.

В более общем случае можно ставить в соответствие множество приоритетов в виде действительных чисел $\{\pi_i^{real}\}$, при этом π_i^{real} – это действительно число, характеризующее приоритет задачи τ_i. В дальнейшем будет предполагаться, что чем меньше значение π_i^{real}, тем выше приоритет задачи τ_i, то есть приоритет задачи τ_i выше приоритета задачи τ_v,

если и только если $\pi_i^{real} < \pi_v^{real}$. Также будет предполагаться, что все приоритеты являются различными.

Пусть $\mathbf{\Gamma}_\pi$ – это список всех задач из множества $\mathbf{\Gamma} = \{\tau_1, \ldots, \tau_n\}$, упорядоченный согласно приоритетам этих задач в порядке снижения приоритетов, то есть на первом месте этого списка находится задача с самым высоким приоритетом.

Пусть $\pi(i)$ – это номер места задачи τ_i в списке $\mathbf{\Gamma}_\pi$. При этом сходство обозначений π_i^{real} и $\pi(i)$ может обосновываться тем, что для любой задачи τ_i значение $\pi(i)$, очевидно, может рассматриваться в качестве альтернативного варианта приоритета π_i^{real} данной задачи. Отличие между π_i^{real} и $\pi(i)$ состоит только в том, что с одной стороны значения π_i^{real} являются действительными числами, и для любых i, v справедливо условие $\pi_i^{real} - \pi_v^{real} \in [-\infty, \infty]$, а с другой стороны значения $\pi(i)$ являются натуральными числами, и для любого i справедливо условие $\pi(i) \in [1, n]$.

Пусть $I(v)$ – это *индекс задачи ЖРВ, имеющей приоритет равный* v. Тогда получается, что $\tau_{I(1)}$ – задача с самым высоким приоритетом; $\tau_{I(n)}$ – задача с самым низким приоритетом; $\tau_{I(\pi(i)-1)}$ – задача, которая является следующей после τ_i в порядке повышения приоритетов в случае, когда $\pi(i) > 1$.

Пусть $\mathbf{I}(\alpha, \beta)$ – это *множество индексов задач ЖРВ, имеющих приоритеты в диапазоне* $[\alpha, \beta]$, т. е. $\mathbf{I}(\alpha, \beta) = \{I(v) \mid \alpha \le v \le \beta\}$. При этом, очевидно, что если $\alpha > \beta$, то $\mathbf{I}(\alpha, \beta) = \varnothing$. Пусть $\mathbf{I}_{hp}(i)$ – *множество индексов задач ЖРВ с приоритетами выше, чем у задачи* τ_i, т. е. $\mathbf{I}_{hp}(i) = \mathbf{I}(1, \pi(i) - 1)\}$. Пусть $hp(i)$ – *множество задач ЖРВ с приоритетами выше, чем у задачи* τ_i, т. е. $hp(i) = \{\tau_v \mid v \in \mathbf{I}_{hp}(i)\}$. Пусть $\mathbf{I}_{lp}(i)$ – *мно-*

жество индексов задач ЖРВ с приоритетами ниже, чем у задачи τ_i, т. е. $\mathbf{I}_{lp}(i) = \mathbf{I}(\pi(i)+1, n)\}$. Пусть $lp(i)$ *– множество задач ЖРВ с приоритетами ниже, чем у задачи* τ_i, т. е. $lp(i) = \{\tau_v \mid v \in \mathbf{I}_{lp}(i)\}$.

В итоге можно считать, что каждая задача τ_i определяется следующими параметрами: признак периодической или спорадической задачи, ограничение ЖРВ, O_i, T_i, π_i, C_i^{Lo}, C_i^{Up}, а также, возможно, другая информация о временных характеристиках запросов.

1.3.6.3. Диспетчеризация

При диспетчеризации выполняется обслуживание очереди запросов.

Основная очередь – это очередь, в которой находятся запросы, ожидающие доступ к вычислительному ресурсу. Они ожидают начало выполнения или возобновление выполнения после прерывания. В один конец очереди поступают запросы, а с другого конца очереди запросы получают доступ к вычислительному ресурсу.

Можно считать, что диспетчеризация полностью обеспечивается тремя алгоритмами, которые можно описать следующим образом.

1) *Алгоритм формирования запросов задач* $\{\tau_i\}$. Этот алгоритм добавляет запрос $\tau_{i,j}$ в основную очередь, если текущий момент времени равен значению $r_{i,j}$. В свою очередь значение $r_{i,j}$ либо вычисляется согласно (1.3), если τ_i – это периодическая задача, либо определяется поступлением информации о внешнем событии, если τ_i – это спорадическая задача. Таким образом, можно говорить о том, что запрос $\tau_{i,j}$ формируется данным алгоритмом. Но в дальнейшем для удобства также будут использоваться выражения вида «запрос, сформированный задачей τ_i». При этом надо понимать, что запросы формируются указанным алгоритмом.

2) *Алгоритм планирования апериодических запросов.* Этот алгоритм добавляет в основную очередь апериодические запросы, назначая каждому из них определенный приоритет. При этом возможен вариант, при котором возникшие апериодические запросы не сразу поступают в основную очередь, а сначала поступают в специальную очередь апериодических запросов, поддерживаемую данным алгоритмом с применением некоторой дисциплины обслуживания. Главная особенность алгоритма состоит в том, что он должен обеспечивать приемлемое качество выполнения апериодических запросов согласно заданному критерию, при этом ограничения ЖРВ, связанные с выполнением запросов задач $\{\tau_i\}$ не должны нарушаться.

3) *Алгоритм обслуживания основной очереди.* Этот алгоритм обеспечивает перемещение запросов в основной очереди и доступ к вычислительному ресурсу согласно приоритетам запросов. На каждом шаге ϵ запрос с наивысшим приоритетом в основной очереди получает доступ к вычислительному устройству и начинает выполняться, если вычислительное устройство простаивает или выполняется запрос с меньшим приоритетом. В последнем случае выполнение запроса с меньшим приоритетом прерывается, и он возвращается в основную очередь. Если же в основной очереди имеется несколько запросов с наивысшим приоритетом, то преимущество при получении доступа к вычислительному устройству получает наиболее ранний из этих запросов. Когда запрос, получивший доступ к вычислительному ресурсу, завершает выполнение, этот запрос удаляется, и доступ к вычислительному ресурсу освобождается для других запросов.

Следует отметить, что 1-й и 3-й алгоритмы реализуются вполне очевидным образом. Что касается 2-го алгоритма, то здесь надо учитывать наличие множества подходов к его реализации. Соответственно разработано множество алгоритмов планирования апериодических запросов, ориентированных на различные критерии качества выполнения апериодических запросов, а также имеющих различные характеристики.

Описания концепций и алгоритмов планирования апериодических запросов можно найти в работах [35, 37, 46, 56, 55, 79, 81, 90, 92].

Таким образом, диспетчеризация, основанное на трех указанных алгоритмах, обеспечивает выполнение запросов при разделении между ними процессорного времени вычислительного ресурса согласно приоритетам этих запросов.

1.3.6.4. Планирование

Важно отметить, что в ходе диспетчеризации ограничения ЖРВ непосредственно не могут учитываться при формировании запросов задач $\{\tau_i\}$ и обслуживании этих запросов в основной очереди. Действительно, при выполнении алгоритма формирования запросов задач $\{\tau_i\}$ учитываются только значения O_i, T_i для каждой периодической задачи или информация о внешних событиях для каждой спорадической задачи. При выполнении алгоритма обслуживания основной очереди принимается во внимание только множество $\{\pi_i\}$. Таким образом, учитывается только множество $\{O_i, T_i, \pi_i \mid i \in [1, n]\}$.

Пусть множество $\{O_i, T_i, \pi_i \mid i \in [1, n]\}$ для краткости обозначается $OT\pi$.

Множество $OT\pi$ порождает некоторую совокупность возможных вариантов выполнения запросов задач $\{\tau_i\}$. Различия между вариантами определяются: возможными вариациями значений $C_{i,j}$; различными последовательностями внешних событий, соответствующих спорадическим задачам; различными последовательностями апериодических запросов. Если в случае каждого из этих вариантов соблюдаются все ограничения ЖРВ задач $\{\tau_i\}$, то при данном множестве $OT\pi$ *гарантируется* соблюдение всех ограничений ЖРВ задач $\{\tau_i\}$.

Определение 1.1. Задача τ_i является *выполнимой*, если гарантируется соблюдение ограничения РВ этой задачи при диспетчеризации.

Кроме обеспечения выполнимости каждой задачи, необходимо также гарантировать ограниченность размера основной очереди, что требуется для практической реализации подсистемы планирования. С учетом этого формулируется следующее определение.

Определение 1.2. Множество $\{\tau_i\}$ является *выполнимым*, если, во-первых, обеспечивается выполнимость каждой задачи из $\{\tau_i\}$, во-вторых, гарантируется, что размер основной очереди всегда будет меньше некоторого конечного значения.

На длительность запросов нельзя повлиять средствами подсистемы планирования, поэтому очевидно, что ограниченность размера очереди можно обеспечить только выбором значений $\{T_i \mid i \in [1,n]\}$.

Тогда понятно, что для обеспечения выполнимости $\{\tau_i\}$ достаточно сформировать подходящие значения O_i, T_i для каждой периодической задачи и сформировать подходящее множество $\{\pi_i\}$.

Формирование запросов спорадической задачи определяется не значениями O_i, T_i, а внешними событиями, но в дальнейшем для краткости будет упоминаться формирование значений O_i, T_i для каждой τ_i, в том числе и спорадической. При этом, если τ_i – это спорадическая задача, то предполагается, что $O_i = 0$, и T_i – это заранее известный минимальный период формирования запросов данной задачи.

Таким образом, до начала функционирования системы надо сформировать множество $OT\pi$, обеспечивающее выполнимость $\{\tau_i\}$, или прийти к выводу, что ни одного такого множества найти не удается. Именно для решения этой проблемы и предназначено планирование.

Сложность проблемы зависит от возможных видов ограничений, т. е. от *целевого класса* ограничений. В частности, если в качестве ограничений могут быть ограничения на интервалы между соседними выполнениями запросов задачи, то тогда указанная проблема является NP-трудной [1, 99]. В подобных случаях практическим решением будет эвристический алгоритм планирования. Конкретная реализация такого алгоритма зависит от целевого класса ограничений, т. е. от вида имеющихся ограничений РВ. Поэтому для дальнейшего рассмотрения планирования надо выделить хотя бы один вид ограничений РВ.

1.3.6.5. Стандартное ограничение реального времени

Обычно (например, см. [44]) стандартное требование к задаче τ_i – это совокупность условий:

1) каждое значение $r_{i,j}$ ограничено соотношением (1.3), когда τ_i – это периодическая задача, или соотношением (1.4), когда τ_i – это спорадическая задача;

2) каждое значение $f_{i,j}$ ограничено условием

$$f_{i,j} \le d_{i,j} = r_{i,j} + D_i \ , \qquad (1.5)$$

где $d_{i,j}$ – *крайний срок* запроса $\tau_{i,j}$, т. е. максимально допустимое время завершения запроса $\tau_{i,j}$; D_i – *относительный крайний срок* задачи τ_i.

На основе стандартного требования можно получить *стандартное ограничение* (СО).

В случае спорадической задачи значения $r_{i,j}$ формируются внешними событиями, и соотношение (1.4) характеризует свойства потока внешних событий. Поэтому (1.4) – это скорее одна из характеристик внешней среды, а не ограничение РВ. Условие (1.5) в случае спорадической задачи определяет допустимое запаздывание при выполнении запроса, соответст-

вующего очередному внешнему событию. Тогда можно сформулировать следующее определение.

СО спорадической задачи τ_i – это условие (1.5), которое должно выполняться при $\forall j \leq 1$.

При этом единственный параметр D_i полностью задает СО спорадической задачи.

Если в случае периодической задачи τ_i в качестве СО использовать совокупность условий (1.3) и (1.5), то тогда возникает следующая проблема. С одной стороны, с помощью (1.5) ограничение накладывается на $f_{i,j}$. Часто предполагается, что момент $f_{i,j}$ связан с некоторым воздействием системы на внешнюю среду, например, в этом момент формируется управляющее воздействие на исполнительное устройство или отправляется некоторая команда по каналу связи. Тогда, как и ожидается, ограничение определяет некоторые правила поведения системы во внешней среде. С другой стороны, с помощью (1.3) ограничение накладывается на $r_{i,j}$. Если τ_i – это периодическая задача, то тогда значение $r_{i,j}$ определяется внутри системы в результате работы подсистемы планирования с учетом используемой концепции планирования. Более того, например, в случае табличного планирования (см. п. 1.3.4.1) вообще нет необходимости отдельно формировать $r_{i,j}$, так как согласно заранее составленному расписанию для каждого очередного запроса периодической задачи сразу же формируется $s_{i,j}$. Тогда оказывается, что ограничение привязано к определенным концепциям планирования, которые предполагают формирование значения $r_{i,j}$. Но этого не должно быть, ведь планирование – это лишь инструмент для того, что обеспечить соблюдение заранее определенных ограничений РВ.

Получается, совокупность условий (1.3) и (1.5) нельзя использовать в качестве СО периодической задачи. Тогда предлагается следующее определение.

СО периодической задачи τ_i – это условие

$$\begin{cases} s_{i,j} \geq O_i^* + (j-1)T_i^* \\ f_{i,j} \leq O_i^* + (j-1)T_i^* + D_i \end{cases}, \qquad (1.6)$$

где O_i^*, T_i^* – параметры ограничения, при этом предполагается, что $O_i^* \geq 0$, $T_i^* > 0$, при этом данное условие должно выполняться при $\forall j \geq 1$.

Условие (1.6), определяющее ограничение РВ, указано в сокращенной форме, так как подразумевается, что (1.6) должно выполняться при $\forall j \geq 1$.

В дальнейшем каждое условие, определяющее ограничение РВ, также будет указываться в сокращенной форме, т. е. всякий раз будет подразумеваться, что это условие должно выполняться при $\forall j \geq 1$.

Важно, что в условии (1.6) нет параметров O_i, T_i задачи τ_i, а есть только параметры СО, обозначаемые O_i^*, T_i^*, т. е. происходит разделение параметров задачи и параметров СО. При этом получается, что СО не связано с какой-либо концепцией планирования. Тогда СО (1.6) может рассматриваться как частный случай некоторого обобщенного ограничения РВ.

Следующее утверждение помогает в решении проблемы выбора значений O_i, T_i, обеспечивающих соблюдение СО (1.6).

Утверждение 1.1. Если СО (1.6) соблюдается при $\forall j \geq 1$ для некоторых O_i, T_i, то оно будет соблюдаться и при $O_i = O_i^*$, $T_i = T_i^*$.

Доказательство. Пусть СО (1.6) задачи τ_i соблюдается при $\forall j \geq 1$ для некоторых O_i, T_i. В ходе выполнения задачи τ_i при $\forall j \geq 1$ возможен один

из трех вариантов: $r_{i,j} < O_i^* + (j-1)T_i^*$, $r_{i,j} = O_i^* + (j-1)T_i^*$, $r_{i,j} > O_i^* + (j-1)T_i^*$.

Пусть при некотором j реализуется вариант $r_{i,j} < O_i^* + (j-1)T_i^*$. Так как СО (1.6) задачи τ_i соблюдается при данном j, то $s_{i,j} \geq O_i^* + (j-1)T_i^*$. Это возможно только в случае, когда выполнение $\tau_{i,j}$ начинается не раньше момента $O_i^* + (j-1)T_i^*$ из-за выполнения задач с более высокими приоритетами на интервале $[r_{i,j}, O_i^* + (j-1)T_i^*)$. Тогда, очевидно, при данном j условие (1.6) также будет соблюдаться, если запрос $\tau_{i,j}$ появится точно в момент $O_i^* + (j-1)T_i^*$, т. е. при условии, что $r_{i,j} = O_i^* + (j-1)T_i^*$.

Пусть при некотором j реализуется вариант $r_{i,j} > O_i^* + (j-1)T_i^*$. При этом СО (1.6) задачи τ_i соблюдается для данного j. Тогда (1.6) также будет соблюдаться и при $r_{i,j} = O_i^* + (j-1)T_i^*$ для данного j. Действительно, первое неравенство условия (1.6) обязательно выполняется, так как $s_{i,j} \geq r_{i,j}$. Второе неравенство условия (1.6) также выполняется, ведь оно выполняется при $r_{i,j} > O_i^* + (j-1)T_i^*$, и значение $f_{i,j}$ не может увеличиться при уменьшении $r_{i,j}$.

Таким образом, если СО (1.6) задачи τ_i соблюдается при $\forall j \geq 1$ для некоторых O_i, T_i, то оно также будет соблюдаться при $r_{i,j} = O_i^* + (j-1)T_i^*$ для $\forall j \geq 1$, т. е. при $O_i = O_i^*$, $T_i = T_i^*$, учитывая (1.3). Утверждение 1.1 доказано.

Если оказывается, что (1.6) не соблюдается при $O_i = O_i^*$, $T_i = T_i^*$, то согласно утверждению 1.1 не существует других O_i, T_i обеспечивающих

соблюдение (1.6). Поэтому удобно использовать $O_i = O_i^*$, $T_i = T_i^*$ в случае СО (1.6).

Если $T_i > T_i^*$, то невозможно обеспечить соблюдение (1.6) при $\forall j \geq 1$. Действительно, очевидно, что в этом случае всегда найдется такое j, что $O_i + (j-1)T_i > O_i^* + (j-1)T_i^* + D_i$, т. е. $r_{i,j} > O_i^* + (j-1)T_i^* + D_i$. Получается, что при таком j условие (1.6) нарушается, учитывая, что $r_{i,j} < f_{i,j}$. Если $T_i < T_i^*$, и (1.6) соблюдается, то тогда, очевидно, будет происходить неограниченное увеличение размера основной очереди. Поэтому при $T_i < T_i^*$ невозможно обеспечить выполнимость $\{\tau_i\}$, учитывая определение 1.2. Таким образом, остается только вариант $T_i = T_i^*$.

Если $O_i < O_i^*$, $T_i = T_i^*$, и (1.6) соблюдается, то выполнение всех запросов задачи τ_i никак не изменится, если установить $O_i = O_i^*$, $T_i = T_i^*$. Вариант, когда $O_i > O_i^*$, $T_i = T_i^*$, очевидно, эквивалентен переходу к новому более жесткому СО (1.6) с меньшим значением D_i. В этом случае можно считать, что для этого нового СО также устанавливаются $O_i = O_i^*$, $T_i = T_i^*$.

Таким образом, в дальнейшем предполагается, что для периодической задачи, имеющей СО (1.6), всегда устанавливаются параметры O_i, T_i согласно соотношениям $O_i = O_i^*$, $T_i = T_i^*$.

Если $O_i = O_i^*$, $T_i = T_i^*$, то для $\forall j \geq 1$ получается, что $r_{i,j} = O_i^* + (j-1)T_i^*$. Также согласно концепции ПФП для $\forall j \geq 1$ справедливо соотношение $r_{i,j} \leq s_{i,j}$.

Утверждение 1.2. Необходимым и достаточным условием соблюдения СО периодической задачи τ_i при $O_i = O_i^*$, $T_i = T_i^*$ будет условие (1.5).

Доказательство. Если $O_i = O_i^*$, $T_i = T_i^*$, то для $\forall j \geq 1$ получается, что $r_{i,j} = O_i^* + (j-1)T_i^*$. Тогда вторая строка условия (1.6) эквивалентна условию (1.5). Поэтому условие (1.5) является необходимым. Кроме того, согласно концепции ПФП для $\forall j \geq 1$ справедливо соотношение $r_{i,j} \leq s_{i,j}$. Тогда при $O_i = O_i^*$, $T_i = T_i^*$ первая строка условия (1.6) выполняется всегда. Поэтому условие (1.5) является также и достаточным. Утверждение 1.2 доказано.

Таким образом, стандартное требование к периодической задаче, выражающееся в виде условий (1.3) и (1.5) можно считать проявлением СО периодической задачи в случае, когда параметры O_i, T_i устанавливаются согласно соотношениям $O_i = O_i^*$, $T_i = T_i^*$.

1.3.6.6. Планирование при наличии только стандартных ограничений

Множество $OT\pi$, обеспечивающее выполнимость $\{\tau_i\}$, будет называться *допустимым*. Формирование параметров O_i, T_i, π_i для каждой задачи τ_i должно осуществлять так, чтобы сформировать допустимое множество $OT\pi$.

В идеале планирование должно всегда формировать допустимое множество $OT\pi$, если существует хотя бы одно такое множество. Очевидно, при ограничениях РВ произвольного вида такой идеальный вариант невозможен на практике. В частности, единственным способом нахождения допустимого $OT\pi$ может оказаться перебор всех вариантов $OT\pi$ и анализ выполнимости $\{\tau_i\}$ для каждого варианта.

При наличии только СО, планирование упрощается.

Во-первых, это становится понятным с учетом следующего утверждения.

Утверждение 1.3. Если все ограничения РВ являются СО, то для выполнимости $\{\tau_i\}$ достаточно, чтобы гарантировалось соблюдение всех СО задач из $\{\tau_i\}$ при диспетчеризации, и для каждой периодической задачи τ_i выполнялись соотношения $O_i = O_i^*$, $T_i = T_i^*$.

Доказательство. Если *спорадическая* задача τ_i имеет СО (1.5), которое соблюдается при $\forall j \geq 1$, то время пребывания в основной очереди каждого запроса $\tau_{i,j}$ ограничено сверху значеним D_i. Учитывая (1.4), ясно, что за время D_i в основной очереди может появиться ограниченное количество запросов задачи τ_i. Таким образом, количество запросов спорадической задачи τ_i в основной очереди в любой момент времени ограничено сверху некоторым фиксированным числом, если соответствующее СО соблюдается.

Если *периодическая* задача τ_i имеет СО (1.6), которое соблюдается при $\forall j \geq 1$, а также имеет $O_i = O_i^*$, $T_i = T_i^*$, то тогда, очевидно, $r_{i,j} = O_i^* + (j-1)T_i^*$ и $f_{i,j} \leq O_i^* + (j-1)T_i^* + D_i = r_{i,j} + D_i$. Это означает, что время пребывания в основной очереди каждого запроса $\tau_{i,j}$ ограничено сверху значеним D_i. Учитывая (1.3), ясно, что за время D_i в основной очереди может появиться ограниченное количество запросов задачи τ_i. Таким образом, количество запросов периодической задачи τ_i в основной очереди в любой момент времени ограничено сверху некоторым фиксированным числом, если соответствующее СО соблюдается, и $O_i = O_i^*$, $T_i = T_i^*$.

Множество $\{\tau_i\}$ содержит только спорадические и периодические задачи. Тогда если все эти задачи имеют только СО, при этом все СО соблюдаются, и для каждой периодической задачи τ_i выполняются соотношения $O_i = O_i^*$, $T_i = T_i^*$, то тогда количество запросов каждой задачи из

$\{\tau_i\}$ в основной очереди всегда ограничено некоторым фиксированным числом, т. е. гарантируется, что размер основной очереди всегда будет меньше некоторого конечного значения. Учитывая определения 1.1, 1.2, получается, что выполняются оба условия, обеспечивающие выполнимость $\{\tau_i\}$. Утверждение 1.3 доказано.

Во-вторых, при наличии только СО можно резко сократить область поиска допустимого $OT\pi$, установив $O_i = O_i^*$, $T_i = T_i^*$ для каждой τ_i. Тогда, если ни одно из возможных множеств $\{\pi_i\}$ не обеспечивает выполнимость, то, учитывая утверждение 1.1, это означает, что в данном случае не существует ни одного допустимого $OT\pi$. Таким образом, в случае СО (1.6) проблема формирования допустимого $OT\pi$ сводится к нахождению подходящего множества $\{\pi_i\}$.

Выбор подходящего множества $\{\pi_i\}$, т. е. назначение приоритетов задачам, выполняет *алгоритм назначения приоритетов*.

Алгоритм назначения приоритетов называется *оптимальным*, если выполнимость любого заданного $\{\tau_i\}$ не обеспечивается данным алгоритмом, только когда не существует $\{\pi_i\}$, обеспечивающего выполнимость $\{\tau_i\}$ [17].

Известно, что, когда каждая задача τ_i имеет СО, а также $O_i = 0$, $D_i \le T_i$ для $\forall i \in [1, n]$, оптимальным алгоритмом назначения приоритетов является алгоритм DM (Deadline Monotonic) [16]. В результате его выполнения получается такое множество $\{\pi_i\}$, что для любых двух задач τ_i и τ_v при условии, что $\pi_i < \pi_v$, всегда справедливо соотношение $D_i \le D_v$. Алгоритм DM сводится к сортировке задач по возрастанию значений D_i и назначению приоритетов, равных порядковым номерам задач в отсортированном списке. Затем, остается выполнить анализ выполнимости $\{\tau_i\}$. Если выполнимость не обеспечивается, то это означает, что ни одного допус-

тимого $OT\pi$ не существует, так как алгоритм DM является оптимальным в данных условиях. Таким образом, планирование заканчивается либо с результатом в виде допустимого $OT\pi$, либо с сообщением о невозможности обеспечить выполнимость $\{\tau_i\}$.

Если при наличии только СО не соблюдается условие, состоящее в том, что $O_i = 0$, $D_i \le T_i$ для $\forall i \in [1,n]$, то алгоритм DM перестает быть оптимальным. Однако в работе [17] был представлен алгоритм, который является оптимальным алгоритмом назначения приоритетов, когда каждая задача τ_i имеет СО, а также значения O_i, T_i, D_i могут быть любыми для $\forall i \in [1,n]$. Этот алгоритм будет воспроизводиться здесь без изменения основного принципа, но в несколько иной форме, что вызвано, во-первых, способом описания алгоритмов, используемым в настоящей работе, во-вторых, необходимостью согласования этого алгоритма с другими алгоритмами.

Для описания алгоритма потребуется следующее определение.

Алгоритм анализа выполнимости задачи τ_i – это алгоритм, который: 1) имеет в качестве исходных данных множество $\{\tau_i\}$, а также индекс i для выделения задачи τ_i в этом множестве; 2) выдает в качестве результата информацию о том, обеспечивается ли выполнимость задачи τ_i.

Теперь можно перейти к описанию указанного алгоритма.

Алгоритм 1.1. Алгоритм назначения приоритетов, который является вариантом представления алгоритма, предложенного в работе [17].

Входные данные алгоритма – это множество $\{\tau_i\}$, а также алгоритмы анализа выполнимости для каждой задачи из $\{\tau_i\}$.

Результат выполнения алгоритма – это либо множество $\{\pi_i\}$ и сообщение о том, что ограничения РВ всех задач из $\{\tau_i\}$ гарантированно со-

блюдаются, либо сообщение о невозможности гарантировать соблюдение ограничений РВ всех задач из $\{\tau_i\}$.

Последовательность шагов алгоритма определяется следующим образом.

1. Устанавливается $\pi_i = i$ для $\forall i \in [1,n]$.
2. Устанавливается $v = n$, где v –определяет текущий назначаемый приоритет.
3. Если $v < 1$, то алгоритм завершается с результатом, содержащим множество $\{\pi_i\}$ и сообщение о том, что ограничения РВ всех задач из $\{\tau_i\}$ гарантированно соблюдаются.
4. Устанавливается $w = I(v)$, $z = I(v)$, где w –определяет индекс очередной задачи среди задач, которым еще не назначен окончательный вариант приоритета; z –сохраняет индекс задачи, имевшей приоритет v, к началу шага 4.
5. Если $w \neq z$, то устанавливается $\pi_z = \pi_w$, $\pi_w = v$. При этом получается, что множество $\mathbf{I}_{hp}(w) \cup \{w\}$ состоит из индексов всех задач, которым еще не назначен окончательный вариант приоритета.
6. Выполняется алгоритм анализа выполнимости, относящийся к задаче τ_w. Если в результате этого выполнения гарантируется выполнимость τ_w, то осуществляется переход к шагу 10.
7. Если $\pi_z = 1$, то алгоритм завершается с результатом в виде сообщения о невозможности гарантировать соблюдение ограничений РВ всех задач из $\{\tau_i\}$, так как уже проверены все задачи, которым еще не назначен окончательный вариант приоритета, и ни для одной из этих задач невозможно гарантировать выполнимость при назначении приоритета, равного v.

8. Если $w \neq z$, то устанавливается $\pi_w = \pi_z$, $\pi_z = v$. При этом получается, что восстановлено распределение приоритетов, имевшееся к началу шага 4.
9. Устанавливается $w = I(\pi_w - 1)$ и выполняется переход к шагу 5.
10. Устанавливается $v = v - 1$ и выполняется переход к шагу 3.

Описание алгоритма 1.1 закончено.

Для формирования множества $\{\pi_i\}$ алгоритму 1.1 надо не более, чем $n(n+1)/2$ раз выполнять анализ выполнимости. Очевидно, что это существенно более эффективно, чем осуществлять перебор всех возможных вариантов назначения приоритетов, число которых равно $n!$, и для каждого варианта выполнять проверку выполнимости множества $\{\tau_i\}$.

Для использования алгоритма 1.1 надо уметь определять алгоритм анализа выполнимости применительно к каждой задаче. Следующее определение играет важную роль в решении этой проблемы.

Определение 1.3. *Условие выполнимости* задачи τ_i – это условие, которое является достаточным для обеспечения выполнимости τ_i, т. е. обеспечения гарантированного соблюдения ограничения РВ задачи τ_i при диспетчеризации.

Вполне очевидно, учитывая условие (1.5) и утверждение 1.2, что в случае СО для спорадической или периодической задачи, а также при $O_i = O_i^*$, $T_i = T_i^*$ в случае периодической задачи, условие выполнимости задачи τ_i – это условие

$$Up(f_{i,j} - r_{i,j}) \le D_i, \qquad (1.7)$$

где $Up(f_{i,j} - r_{i,j})$ – это верхняя оценка значений $f_{i,j} - r_{i,j}$ для $\forall j \ge 1$, при этом известны различные способы вычисления $Up(f_{i,j} - r_{i,j})$ [51, 60, 59, 95, 93].

Согласно определению 1.3 условие выполнимости является достаточным условием того, что ограничение РВ для τ_i не будет нарушено в ходе выполнения τ_i. но оно совсем не обязательно является также необходимым условием этого. Поэтому условие выполнимости может быть более или менее точным в смысле количества отвергаемых вариантов, которые, на самом деле, гарантируют соблюдение ограничения РВ. Например, условия $Up(f_{i,j} - r_{i,j}) \le D_i / 2$, $Up(f_{i,j} - r_{i,j}) \le D_i / 3$, … , $Up(f_{i,j} - r_{i,j}) \le D_i / k$, где $k > 0$, также являются условиями выполнимости τ_i в случае СО. Однако, очевидно, что более точным является условие $Up(f_{i,j} - r_{i,j}) \le D_i$, и именно его выгоднее применять на практике. В свою очередь точность этого условия зависит от точности оценки $Up(f_{i,j} - r_{i,j})$, ведь использование подобной оценки в условии (1.7) делает это условие достаточным, но не необходимым.

Теперь легко получить следующий вполне очевидный алгоритм.

Алгоритм 1.2. Алгоритм анализа выполнимости задачи τ_i, имеющей СО.

Входные данные алгоритма – это множество $\{\tau_i\}$, а также индекс i для выделения задачи τ_i в этом множестве.

Результат выполнения алгоритма – это информация о том, гарантируется ли выполнимость задачи τ_i.

Последовательность шагов алгоритма определяется следующим образом.

1. С учетом параметров задачи τ_i, а также параметров задач из $hp(i)$ вычисляется значение $Up(f_{i,j} - r_{i,j})$.
2. Если условие выполнимости (1.7) выполняется, то алгоритм завершается с результатом, что выполнимость τ_i гарантируется, иначе алгоритм завершается с результатом, что выполнимость τ_i не гарантируется.

Описание алгоритма 1.2 закончено.

Таким образом, при наличии только ограничений СО формирование множества $OT\pi$ сводится к установке $O_i = O_i^*$, $T_i = T_i^*$ для каждой τ_i и использованию алгоритма 1.1 для формирования множества $\{\pi_i\}$ с учетом алгоритма 1.2. Более формально этот подход можно представить в виде следующего алгоритма.

Алгоритм 1.3. Планирование при наличии только СО.

Входные данные алгоритма – это множество $\{\tau_i\}$ с неопределенными значениями $\{O_i, T_i, \pi_i \mid i \in [1,n]\}$.

Результат выполнения алгоритма – это либо множество $\{\tau_i\}$ с определенными значениями $\{O_i, T_i, \pi_i \mid i \in [1,n]\}$ и сообщение о выполнимости $\{\tau_i\}$, либо сообщение о невозможности обеспечить выполнимость $\{\tau_i\}$.

Последовательность шагов алгоритма определяется следующим образом.

1. Для каждой периодической задачи τ_i устанавливается $O_i = O_i^*$, $T_i = T_i^*$. В результате формируется множество $\{O_i, T_i \mid i \in [1,n]\}$.
2. Выполняется алгоритм 1.1, при этом в качестве его входных данных используются: множество $\{\tau_i\}$, а также алгоритм 1.2 для каждой задачи из $\{\tau_i\}$. Если в результате этого выполнения оказывается, что множество $\{\pi_i\}$ сформировано, и ограничения РВ всех задач из $\{\tau_i\}$ гарантированно соблюдаются, то тогда алгоритм завершается с результатом, содержащим множество $\{\tau_i\}$ с определенными значениями $\{O_i, T_i, \pi_i \mid i \in [1,n]\}$ и сообщение о выполнимости $\{\tau_i\}$, иначе алгоритм завершается с сообщением о невозможности обеспечить выполнимость $\{\tau_i\}$.

Описание алгоритма 1.3 закончено.

Утверждение 1.4. При наличии только СО множество $\{\tau_i\}$ является выполнимым, если алгоритм 1.3 с входными данными в виде $\{\tau_i\}$ выдает сообщение о выполнимости $\{\tau_i\}$.

Доказательство. Учитывая обоснование оптимальности алгоритма, представленного в работе [17] и соответствующего алгоритму 1.1, а также учитывая вполне очевидный алгоритм 1.2, ясно, что сообщение о выполнимости $\{\tau_i\}$ означает гарантированное соблюдение всех СО. Тогда для выполнимости $\{\tau_i\}$, учитывая определения 1.1, 1.2, остается доказать, что размер основной очереди будет ограничен. Но это следует из утверждения 1.3 с учетом того, что в результате шага 1 алгоритма 1.3 для каждой периодической задачи устанавливается $O_i = O_i^*$, $T_i = T_i^*$. Утверждение 1.4 доказано.

1.4. Современное состояние научных исследований, связанных с планированием задач реального времени

1.4.1. Пример планирования для заданного множества задач

На рис. 11 показан пример выполнения задач реального времени. При этом, заштрихованным прямоугольником обозначается выполнение запроса, окружностью – момент формирования запроса (его активации), сплошной линией – ожидание запроса в очереди. Стрелкой, направленной вниз, обозначается крайний срок выполнения запроса, при этом пунктирная линия, соединяющая окружность и стрелку, показывает разрешенный интервал запроса.

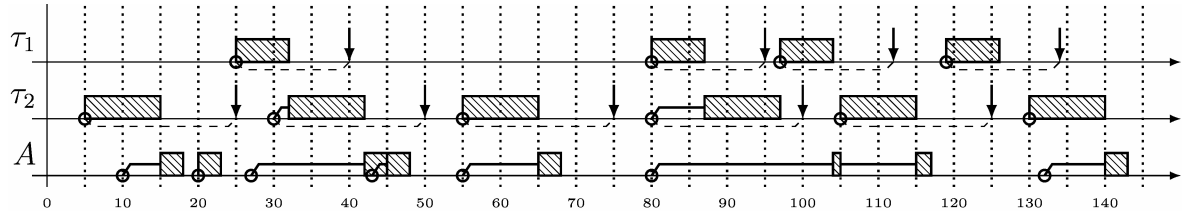

Рис. 11. Пример выполнения задач реального времени

В этом примере τ_1 – спорадическая задача с параметрами: $T_1 = 17$, $C_1 = 7$, $D_1 = 15$, а τ_2 – периодическая задача с параметрами: $O_2 = 5$, $T_2 = 25$, $C_2 = 10$, $D_2 = 20$. Апериодические запросы, обозначаемые A, постоянной длительности, равной 3, формируются в некоторые (заранее не определенные) моменты времени. Здесь в ходе решения проблемы планирования, задаче τ_1 был назначен наивысший приоритет. При этом запросы из потока A выполняются в фоновом режиме, то есть когда процессор свободен от τ_1 и τ_2.

Если задаче τ_2 присвоить наивысший приоритет, то тогда запрос задачи τ_1 не будет укладываться в свой крайний срок при прерывании запросом задачи τ_2. Поэтому, представленное на рис. 11 распределение приоритетов между задачами τ_1 и τ_2 – единственно допустимое. С помощью рис. 11 можно понять гарантированность соблюдения ограничений (1.5) для τ_1 и τ_2 при указанном распределении приоритетов. Но в общих и более сложных случаях, требуется выполнение теста выполнимости, например, на основе алгоритма из работы [17].

Также из этого примера видно, что при уменьшении минимального периода наступления внешний событий, связанных с задачей τ_1 (значение T_1), будут нарушаться ограничения (1.5) задачи τ_2. Например, если запрос $\tau_{1,4}$ появится не в момент 97, а в момент 96, то тогда произойдет прерывание запроса задачи τ_2, и он выйдет за пределы разрешенного интервала $[80,100)$, то есть будет нарушен крайний срок $d_{2,4} = 100$. Возможным решением в данной ситуации может стать уменьшение времени выполнения

запросов задачи τ_2, например, за счет оптимизации программного кода или использования более быстродействующего процессора.

1.4.2. Развитие стандартных моделей онлайнового планирования

Рассмотренный выше пример модели планирования является расширением наиболее простой модели из этого семейства, а именно модели, представленной еще в 1973 году в работе [57]. Позднее появившаяся спорадическая модель [66] характерна присутствием в ней спорадических задач и независимостью относительного крайнего срока от периода. Потом была предложена мультифреймовая модель (multiframe model) [65], в которой задача формирует запросы с длительностями из некоторого набора значения, а не с одним и тем же значением C_i. Но в этой модели относительные крайние сроки совпадали с периодами. Этот недостаток был позднее решен в обобщенной мультифреймовой модели (generalized multiframe model) [19].

Дальнейшее обобщение модели было основано на том, что порядок запросов с различными длительностями не обязательно представляет собой простую последовательность, а может быть задан направленным ациклическим графом (модель повторяющихся задач) [20], что позволяет моделировать ветвление кода программы. Параллельно, другое обобщение было сделано в нециклической обобщенной мультифреймовой модели [68]. Оно основано на том, что отдельные запросы могут идти в различном порядке, тем самым, нарушая цикличность обобщенной мультифреймовой модели. Два этих подхода были объединены в нециклической модели повторяющихся задач [21]. В работе [85] предложено дальнейшее обобщение в виде модели задач реального времени на основе орграфа.

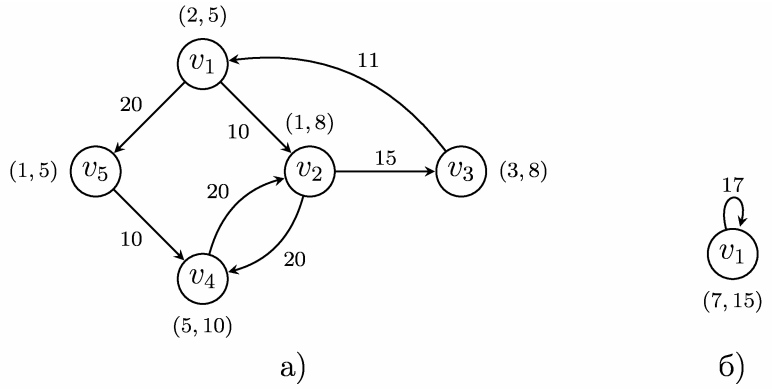

Рис. 12 Примеры орграфов для задач реального времени: а) из работы [85]; б) орграф, соответствующий задаче τ_1 из примера, относящегося к рисунку 11

На рис. . 12 представлен пример орграфа из работы [85], который определяет запросы одной задачи. Этот орграф, как можно видеть, содержит циклы, что расширяет модель из работы [21]. Данный орграф отражает 5 разных вариантов запросов, обозначаемых $v_1,...,v_5$, которые может формировать одна задача τ_i. Для каждой вершины графа, то есть для каждого вида запроса v_i, задается пара чисел, которые определяют соответственно, длительность запроса и относительный крайний срок для данного запроса. Направления ребер графа определяют возможный порядок формирования запросов, а число, соответствующее ребру, определяет минимально возможное расстояние по времени между соседними запросами. Перемещаясь по направлениям ребер этого графа, можно генерировать бесконечную последовательность формирования запросов. Например, можно несколько раз перемещаться по циклу v_1,v_2,v_3 и обратно к v_1, а потом переключиться на цикл v_1,v_5,v_4,v_2,v_3,v_1 и т.д. Разные виды запросов $v_1,...,v_5$ данной задачи могут соответствовать отдельным частям программного кода задачи, которые активируются в зависимости, например, от изменения внутреннего состояния задачи с учетом возникновения некоторых внешних событий.

Для иллюстрации того, насколько данная модель расширяет простую модель планирования, относящуюся к рисунку 11, рассмотрим модель в виде орграфа для задачи τ_1, представленную на рисунку . 12,б. Этот граф

содержит лишь одну вершину и одно ребро, так как в простой модели, относящейся к рисунку 11, задача τ_1 формирует запрос одного типа со значениями длительности $C_1 = 7$ и крайним сроком $D_1 = 15$. При этом минимальный период $T_1 = 17$ отражается на единственном ребре графа.

Естественно, что увеличение выразительности модели, как, например, в случае модели из работы [85], приводит к усложнению теста выполнимости, в частности, из-за усложнения ограничений реального времени и параметров задач реального времени. Хотя бы достаточно сравнить сложности графов на рисунке . 12,а и рисунке . 12,б. При этом, надо учесть, что орграф может быть и еще сложнее. В целом, расширение модели планирования в рамках концепции онлайнового планирования обычно усложняет тест выполнимости.

Модель задач на основе орграфа указывается как наиболее обобщенная из стандартных моделей, обеспечивающих псевдополиномиальную сложность теста выполнимости, среди известных моделей [85]. Также в этой работе отмечено, что, по оценке авторов, они очень близко подошли к пределу псевдополиномиальной сложности, то есть в дальнейшем, будет очень трудно оставаться пределах псевдополиномиальности при увеличении выразительности.

В качестве примера одной из наиболее выразительных моделей можно привести модель автомата задач (task automata) [43]. Но эта модель находится за пределами области псевдополиномиальной сложности. В случае использования модели автомата задач, тест выполнимости может потребовать значительных вычислительных ресурсов. Более того, была показана неразрешимость алгоритмической проблемы построения единого алгоритма для теста выполнимости любой заданной конфигурации задач, описываемой автоматом задач [43].

Даже при наличии более выразительных моделей, более простые мо-

дели также не теряют свою актуальность, так как тесты выполнимости для них обладают меньшей сложностью, и, например, поэтому они удобны для систем с динамическими изменениями множества задач.

1.4.3. Разработка нестандартных моделей планирования задач жесткого реального времени

Существенное количество работ, посвящается разработке моделей планирования задач с ограничениями реального времени, которые отличаются от стандартных ограничений, выражаемых в виде крайних сроков. Рассмотрим несколько примеров таких моделей.

В работе [24] была предложена модель «слабо-жестких» (weakly hard) систем реального времени, в которых разрешается нарушать некоторый, заранее определенный, процент крайних сроков. Это похоже на мягкое реальное время, но принципиальное отличие здесь в том, что заранее гарантируется некоторый минимальный процент соблюдаемых крайних сроков. Тогда можно говорить о жестком ограничении для задачи. Просто это ограничение накладывается не на каждый запрос в отдельности, а на всю совокупность запросов, формируемых данной задачей. В качестве примера использования этой модели можно привести работу [105], в которой «слабо-жесткая» модель рассматривается в контексте дискретно-событийных систем.

Отдельного внимания заслуживает подход, основанный на комбинировании двух основных концепций планирования, в частности, представленный в работах [99, 52]. Общий принцип данного подхода основан на том, что нестандартные ограничения жесткого реального времени гарантируются за счет предварительного составления статического расписания для задач с этими ограничениями, то есть сначала используется планирование на основе расписания. Затем, составленное расписание трансформируется в простую модель онлайнового планирования за счет, например, создания

фиктивных задач. Далее, в полученную модель онлайнового планирования можно добавлять спорадические задачи. Спорадические задачи активируются внешними событиями, поэтому предварительное составление статического расписания для них – это неэффективное решение. В данном случае, во-первых, за счет составления статического расписания гарантируется соблюдение сложных нестандартных ограничений. Во-вторых, обеспечивается хорошая гибкость планирования за счет использования онлайнового планирования на последующей стадии

1.4.4. Исследования на стыке теории планирования и теории автоматического управления

Уже довольно давно было замечено, что стандартная модель периодических задач с крайними сроками не очень эффективна (в смысле использования ресурсов) для реализации задач, обеспечивающих контуры управления. В частности, требование строго периодического выполнения является излишним ужесточением, и очень часто это требование можно ослабить, высвобождая существенные ресурсы. Однако, с другой стороны, алгоритмы управления также предполагают строгую периодичность выполнения с заданным периодом дискретизации. Поэтому необходимы решения, затрагивающие как планирование, так и алгоритмы управления.

Один из ранних подходов на этом направлении был представлен в работе [63]. Согласно данному подходу, вместо ограничения строгой периодичности разрешенных интервалов для запросов, как это делается в стандартной модели периодических задач, было введено понятие гибких ограничений реального времени (flexible timing constraints). Гибкое ограничение определяет допустимый интервал между моментами стартов двух соседних запросов задачи, которая получает измерительные данные, а также допустимый интервал между стартом запроса задачи получения измерительных данных и завершением запроса задачи формирования управ-

ляющего воздействия на объект управления. Тем самым, снимается требование строгой периодичности формирования запросов. Одновременно, алгоритм управления модифицируется на основе компенсационного подхода так, чтобы учитывать нестрогую периодичность формирования запросов. Простейшим вариантом в случае ПИД-регулятора будет динамическое изменение периода дискретизации, значение которого влияет на вычисляемое значение интегральной и дифференциальной составляющих. В итоге, достигается требуемое качество управления при более эффективном использовании вычислительных ресурсов.

Есть подходы, которые продвигаются дальше по пути совместного использования методов теории планирования и теории автоматического управления. Основное преимущество таких подходов в том, что они позволяют повысить эффективность использования ресурсов за счет уменьшения посреднической роли ограничений реального времени, которые обычно и определяют границу между теорией планирования и теорией автоматического управления.

К подобным подходам можно отнести решения, связанные с самоактивирующимися задачами (self-triggered tasks) для контуров управления. В данном случае, частота формирования запросов управляющей задачи зависит от состояния контура управления [97]. Например, можно уменьшать частоту запросов, если объект управления находится в равновесии, а при выходе из равновесия частота запросов будет увеличиваться. Тогда существенным образом можно увеличить ресурсоэффективность [13, 14]. В частности, можно уменьшить среднее энергопотребление управляющего устройства, что особенно важно в случае мобильных устройств.

Но с точки зрения планирования, такой подход требует построения новой модели планирования и решения проблемы достижения заданных критериев качества управления.

В целом, много исследований ведется на стыке планирования и авто-

матического управления применительно к самоактивирующимся задачам, так как самоактивация требует совместного рассмотрения как подсистемы планирования, так и алгоритмов управления. В качестве примера, можно привести работу [100].

1.4.5. Планирование с обратной связью

Как уже было отмечено, если планирование задач, задействованных в контурах управления, осуществлять совместно с проектированием этих контуров управления, то можно добиться более эффективного использования ресурсов. В качестве отдельного направления можно выделить так называемое планирование с обратной связью, которое предоставляет собой еще один интересный способ ослабления посреднической роли ограничений реального времени.

Основная идея планирования с обратной связью состоит в том, что распределение ресурсов между задачами реального времени осуществляется на основе обратной связи, которая сигнализирует о качестве управления, осуществляемого данными задачами. Например, если у одной из задач снижается качество управления, то ей выделяется дополнительное процессорное время, и она может увеличить частоту получения измерительных данных или увеличить длительность своих запросов, подключая более эффективный алгоритм управления.

В монографии [104] систематизировано излагаются основные подходы и проблемы, относящиеся к планированию с обратной связью. В ней же можно найти большой перечень научных работ по данной тематике. В качестве примеров более поздних исследований можно привести работы [23, 98].

Наряду с указанными преимуществами планирования с обратной связью, следует отметить основной недостаток, который состоит в том, что заранее трудно гарантировать определенный уровень качества управления.

В случае обратной связи по качеству управления, актуальным является вопрос разработки методов обеспечения предсказуемости качества управления при использовании планирования с обратной связью.

Однако, несмотря на сложность разработки таких методов, данное направление можно считать одним из наиболее перспективных, так как оно очень далеко продвигается по пути стыковки теории планирования задач реального времени и теории автоматического управления, а также обеспечивает высокую адаптивность к динамическим изменениям за счет обратных связей. Учитывая современные тенденции развития архитектур систем управления, связанные с развитием адаптивности, мобильности и ресурсоэффективности, вполне очевидной становится актуальность концепции планирования с обратной связью.

1.4.6. Совместное планирование для жесткого и мягкого реального времени

Одна из основных тенденций в развитии исследований в области проблем планирования – это разработка моделей и методов планирования, которые позволяют осуществить гарантированное соблюдение жестких ограничений и, вместе с этим, максимально улучшить качество применительно к требованиям мягкого реального времени.

Изначально на данном направлении рассматривались проблемы планирования апериодических запросов на фоне задач жесткого реального времени. Для минимизации среднего времени задержки апериодических запросов (мягкого реального времени) были, в частности, разработаны алгоритмы захвата резерва (slack stealing) [38]. Поясним суть этого подхода на примере. На рисунке 11 апериодические запросы выполняются с низшим приоритетом (в фоновом режиме), поэтому они очень часто выполняются с задержками, и среднее время задержки по первым семи запросам равно (5+0+15+2+10+34+8) / 7 = 10.57 ≈ 11. Если же вместо фонового вы-

полнения использовать алгоритм захвата резерва в «жадном» (greedy) варианте, то тогда запросы будут выполняться, как указано на рис. 13.

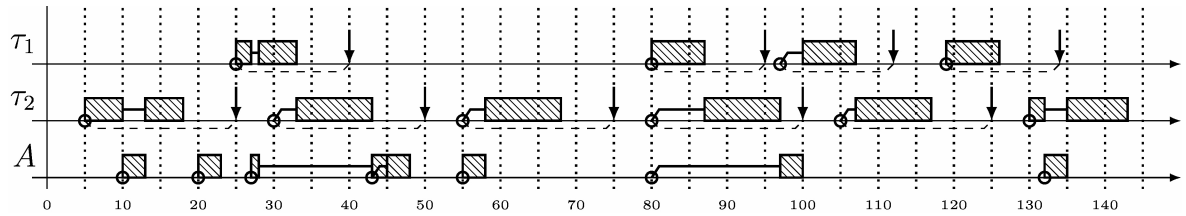

Рис. 13 – Пример выполнения задач реального времени при более эффективном планировании апериодических запросов

В данном случае запросы задач τ_1 и τ_2 «отодвигаются» вправо внутри своих разрешенных интервалов, определяемых значениями крайними сроками, чтобы «пропустить вперед» апериодические запросы. Тем самым, среднее время задержки уменьшается до значения (0+0+15+2+0+17+0) / 7 = 4.86 ≈ 5, то есть примерно в 2 раза. Одновременно с этим, гарантируется соблюдение жестких ограничений (1.5) для τ_1 и τ_2. Это пример повышения эффективности совместного планирования для жесткого и мягкого реального времени.

Дальнейшее развитие этого направления привело к более сложным моделям, в которых у одной задачи может быть и жесткая, и мягкая составляющая ограничения. Тогда метод планирования гарантирует соблюдение некоторого минимально допустимого уровня качества, выражаемого жесткой составляющей ограничения, и стремится по возможности повысить уровень качества, ориентируясь на мягкие составляющие. В качестве представителей такого подхода можно выделить работы по так называемым эластичным задачам (elastic task) [30]. У каждой такой задачи есть максимально допустимый период, выполнение с которым гарантируется (жесткая составляющая). Но, по возможности, система старается делать период поменьше (мягкая составляющая), учитывая различие в «коэффициентах эластичности» задач.

Другой пример – это гравитационная модель задач [50], в которой каждая задача подобно маятнику, который вне особых воздействий (без движения) указывает на целевую точку в разрешенном интервале запроса (между стартом и крайним сроком). Но взаимовлияние задач вынуждает «маятник» смещаться от целевой точки, что приводит «к поднятию маятника» относительно этой точки, где его энергия минимальна. Запрос гарантированно находится в разрешенном интервале (жесткая составляющая), но при этом минимизируется энергия «задач-маятников», что приближает их к желаемой точке внутри интервала (мягкая составляющая).

1.4.7. Специфика применения новых методов планирования в проектах управляющих систем реального времени

Развитие методов планирования задач реального времени имеет длинную историю, в несколько десятков лет. Однако использование результатов исследований при разработке управляющих СРВ сталкивается с определенными сложностями. Есть сложность общего порядка, состоящая в том, что инженерная практика обычно отстает от уровня текущих исследований, и иногда на многие годы. Но есть и специфические сложности.

Первая сложность заключается в том, что новые алгоритмы онлайнового планирования требует поддержки со стороны операционных систем (ОС) реального времени. Поэтому часто между новым разработанным алгоритмом и практическим внедрением возникает барьер в виде ОС, где эти новые алгоритмы должны быть реализованы на системном уровне. Причина в том, что развитие коммерческих ОС в части механизма планирования задач идет довольно медленно. Во многом это связано с тем, что добавление новых функций требует тщательного тестирования и может быть связано с существенными изменениями на уровне архитектуры. Также надо учитывать необходимость следования стандартам. Например, расширение стандарта POSIX реального времени (RT-POSIX) определяет специфика-

цию сервисов реального времени, обеспечивающих планирование только с фиксированными приоритетами, и предполагается, что лишь со временем RT-POSIX будет развиваться в сторону более современных и гибких методик планирования, в частности, планирования с динамическими приоритетами [28].

Вторая сложность связана с тем, что новые методы и алгоритмы оффлайнового планирования (ориентированные на этап проектирования системы) основываются на все более сложных моделях. Поэтому они требуют определенного времени для освоения и принятия в инженерную практику проектирования систем реального времени, которая и без того является очень сложной областью деятельности. Чтобы в этом убедиться, достаточно обратиться к замечательной книге [102], показывающей на конкретных примерах всю сложность разработки программного обеспечения для систем реального времени и разнообразие проблем, которые требуется решать при этом. А проблема планирования – это лишь одна из таких проблем. Но от этого она не становится менее значимой. Вместе с тем, на практике разработчики часто обращаются к старым проверенным (но менее эффективным) решениям проблем планирования, экономя, тем самым, трудозатраты, что в итоге приводит к снижению ресурсоэффективности.

1.5. Контрольные вопросы

1. Что такое система реального времени, вычислительная система реального времени? Можно ли рассматривать систему управления как системы реального времени и почему?
2. Каковы особенности процесса разработки системы реального времени.
3. Что такое задачи реального времени?
4. Каким образом задачи реального времени могут быть представлены на различных уровнях организации программного обеспечения сис-

тем реального времени?

5. Что такое запросы реального времени? Каковы основные состояния запросов при их выполнении?
6. Какие вы знаете основные базовые параметры задач реального времени?
7. Что такое ограничение реального времени, и как оно формируется? Чем отличаются ограничения жесткого реального времени от ограничений мягкого реального времени.
8. В чем заключается проблема планирования задач реального времени?
9. Какие базовые модели планирования можно выделить, и в чем состоят принципиальные различия этих базовых моделей?
10. Каковы основные особенности модели планирования на основе фиксированного расписания?
11. Каковы основные особенности модели планирования с фиксированными приоритетами?
12. Каковы основные особенности модели планирования с динамическими приоритетами?
13. В чем выражаются предсказуемость и гибкость планирования задач реального времени? Насколько предсказуемость и гибкость являются противоречащими друг другу понятиями? Каковы уровни предсказуемости и гибкости различных базовых моделей планирования?
14. Каковы основные направления современных исследований в области планирования задач реального времени систем управления?

2. Разработка и использование системного программного обеспечения управляющих систем реального времени

2.1. Инструментальные средства разработки программного обеспечения управляющих систем реального времени

2.1.1. Применение языков программирования низкого и высокого уровня при разработке управляющих систем реального времени

При разработке программного обеспечения СРВ могут использоваться различные языки программирования и инструментальные средства. Среди языков программирования обычно выделяют два класса языков:

– языки программирования низкого уровня (ассемблер и другие языки, близкие к программированию в машинных кодах);

– языки программирования высокого уровня (языки с достаточным уровнем абстракции, например, Си, Си++, Паскаль, Java, Python и др.).

Низкоуровневые языки программирования обычно используются в тех случаях, когда требуется дополнительная оптимизация ресурсов (процессорного времени, памяти), или когда нет возможности использования языков высокого уровня. В остальных случаях чаще всего используются высокоуровневые языки программирования.

Особо следует отметить язык программирования Си, который является достаточно низкоуровневым, хотя его и следует отнести к классу языков высокого уровня. В частности, это проявляется в том, что он является, пожалуй, самым популярным языком системного программирования. Например, этот язык используется для разработки ядра Linux. Также с помощью него обычно создаются драйверы устройств, что часто бывает необхо-

димым в случае разработки нового технического устройства или при подключении нестандартного устройства. В частности, язык Си также используется в качестве базового в монографии [102], в которой излагаются основные особенности разработки программного обеспечения для систем реального времени. В учебном пособии [4], где рассматриваются особенности разработки программного обеспечения встроенных вычислительных систем, также в качестве базового используется язык программирования Си. Учитывая все это, в дальнейшем в качестве базового будет рассматриваться именно язык программирования Си.

Основы и детали языка программирования Си излагаются в работах [2, 3, 5, 11].

2.1.2. Интегрированные среды разработки

2.1.2.1. Общие сведения

При разработке программного обеспечения важно использовать эффективные инструментальные средства, повышающие качество работы программист. Пожалуй, главным классом таких инструментальных средств являются интегрированные среды разработки, которые, в общем случае, в своем составе содержат (или имеют связи для запуска) следующие основные компоненты:

– компилятор;

– компоновщик;

– средства отладки;

– систему контроля версий.

Также в случае настраиваемых интегрированных сред разработки, можно к ним подключить дополнительные инструментальные средства.

Поэтому важно рассмотреть основы разработки программного обеспечения с помощью интегрированных сред разработки.

2.1.3. Среда разработки Code::Blocks

В качестве примера интегрированной среды разработки будет рассматривать среду Code::Blocks [34].

К основным преимуществам Code::Blocks относятся:

– свободная лицензия GPL v3.0 [48], в частности, разрешается бесплатное распространение и использование;

– среда может работать в операционных системах семейств Windows, Linux, OS X (то есть является кросс-платформенной);

– возможность работы с различными компиляторами.

Кроме того, в среде Code::Blocks можно разрабатывать программы для вычислительных устройств на базе микропроцессоров семейства AVR фирмы Atmel (с использованием комплекта компиляторов GNU AVR GCC), а также для вычислительных устройств на базе микропроцессоров различных семейств, поддерживаемых компилятором SDCC.

В статьях [6, 7] можно найти информацию о настройке и об использовании среды Code::Blocks для разработки AVR-приложений и SDCC-приложений. Все это определяет возможность использования данной среды разработки в случае встроенных систем управления, основанных на микропроцессорах указанных семейств.

Но в рамках данного учебного пособия мы рассмотрим пример использования Code::Blocks для создания программ на обычном компьютере, так как это упрощает начальное знакомство с данной средой разработки. В дальнейшем читатель может обратиться к статьям [6, 7] и попробовать использование Code::Blocks для разработки приложений встроенных систем.

На сайте Code::Blocks [34] в разделе Downloads можно найти информацию и ссылки на файлы для установки свежей версии этой среды разработки применительно к той или иной операционной системе.

Далее приводятся сведения об установке Code::Blocks версии 10.05. На момент написания данного текста эта версия является наиболее свежей

из официально выпущенных.

Для примера в качестве базовой операционной системы будет рассматриваться операционная система семейства Microsoft Windows (XP, Vista, 7).

В случае другой операционной системы, например, в случае Linux можно установить Code::Blocks с помощью пакета для нужного дистрибутива среди перечисленных на этой соответствующей странице официального сайта [34]. Также скорее всего Code::Blocks находится в репозитории используемого дистрибутива Linux.

2.1.3.1. Установка среды разработки

Сначала необходимо скачать файл установки codeblocks-10.05mingw-setup.exe по ссылкам, указанным на странице "Download binary" сайта Code::Blocks [34].

Этот файл устанавливает как саму среду Code::Blocks, так и среду MinGW [64], которая предоставляет компиляторы семейства GCC [47]. Среда MinGW тоже относится к свободно распространяемому программному обеспечению.

Надо запустить файл codeblocks-10.05mingw-setup.exe. Появится окно программы установки (рис. 14).

Рис. 14 – Программа установки (шаг 1)

Надо нажать "Next". Появится окно с текстом свободной лицензии GPL v3.0 (рис. 15).

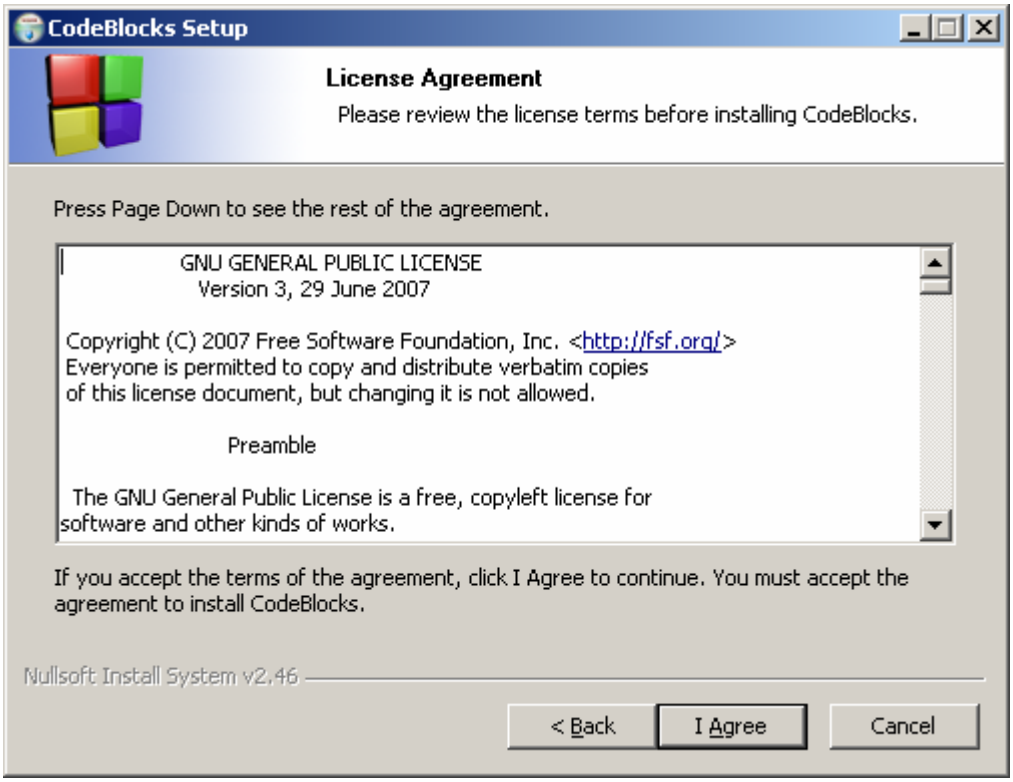

Рис. 15 – Программа установки (шаг 2)

Рис. 16 – Программа установки (шаг 3)

Нажатие кнопки "I Agree" означает согласие с условиями лицензии и позволяет продолжит установку программы (рис. 16), в ходе которой далее появится окно выбора компонентов (рис. 17). В этом окне надо выбрать тип установки "Full: All plugins, all tools, just everything".

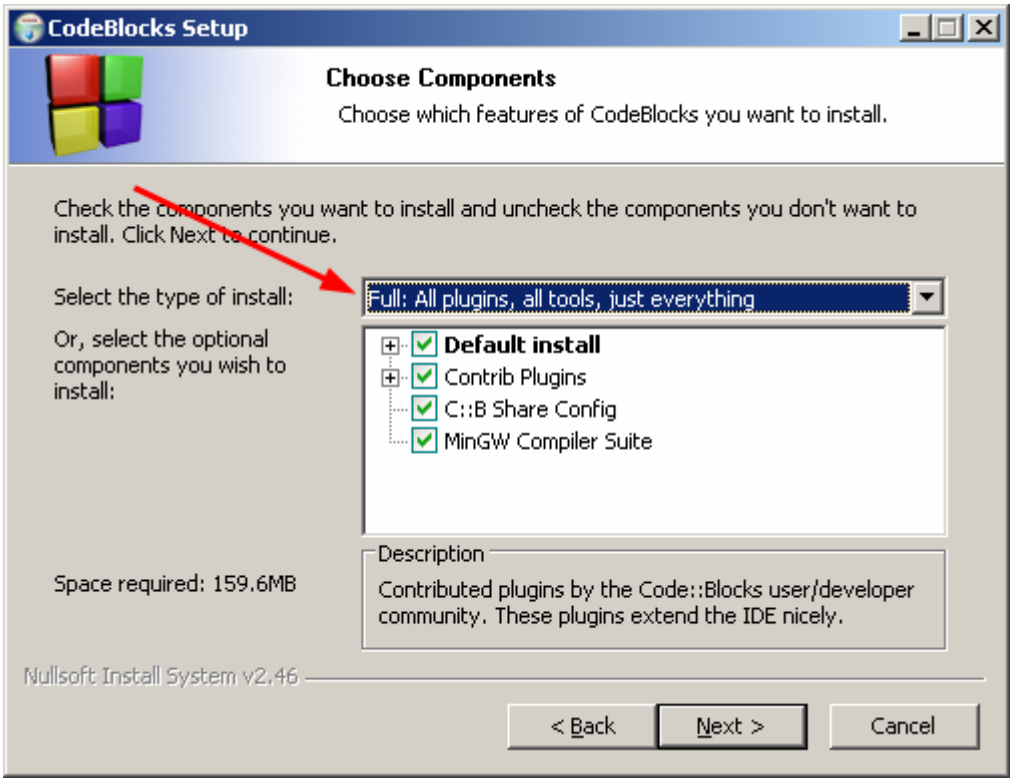

Рис. 17 – Программа установки (шаг 5)

После этого надо нажать кнопку "Next". Появится окно с именем каталога для установки (рис. 18). Лучше, чтобы полное имя каталога не содержало пробелов и других букв, кроме латинских. Поэтому, например, имя C:\Program Files\CodeBlocks лучше заменить на другое, например, на C:\CodeBlocks или C:\Programs\CodeBlocks.

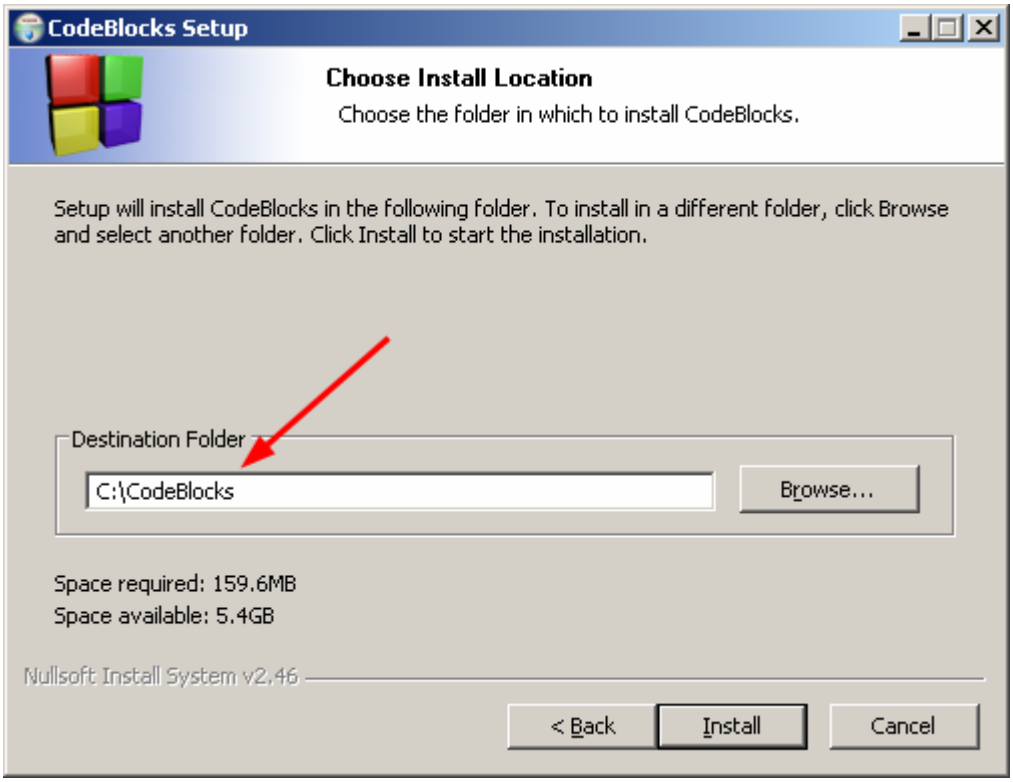

Рис. 18 – Программа установки (шаг 6)

После этого можно нажать кнопку "Install", которая запускает собственно процесс установки (рис. 19).

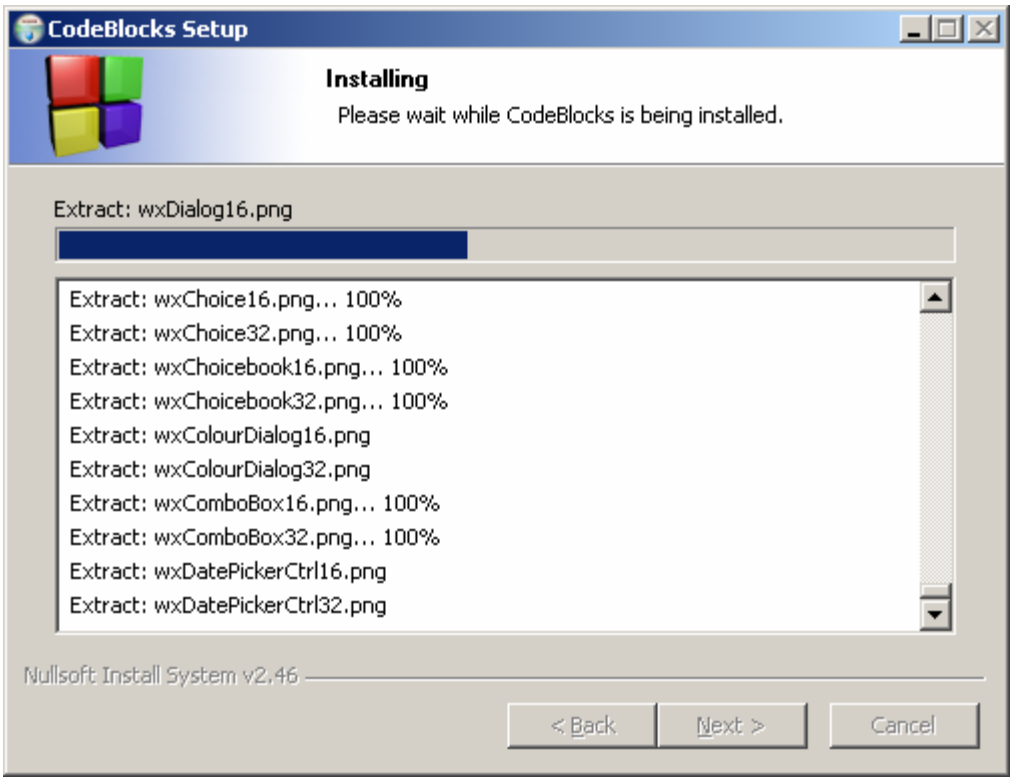

Рис. 19 – Программа установки (шаг 7)

По окончании процесса появится окно с вопросом о запуске Code::Blocks (рис. 20). Лучше здесь нажать "Нет", чтобы потом проверить запуск Code::Blocks обычным путем.

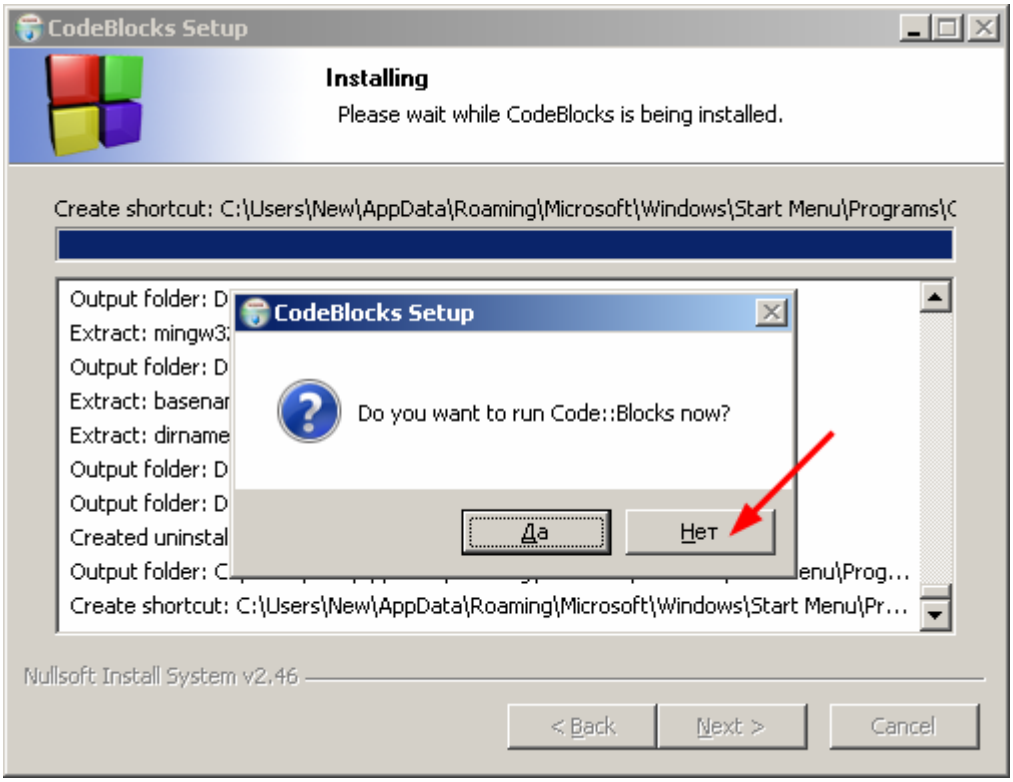

Рис. 20 – Программа установки (шаг 8)

После этого достаточно будет нажать кнопку "Next" (рис. 21), а потом "Finish" (рис. 22), чтобы закончить процесс установки.

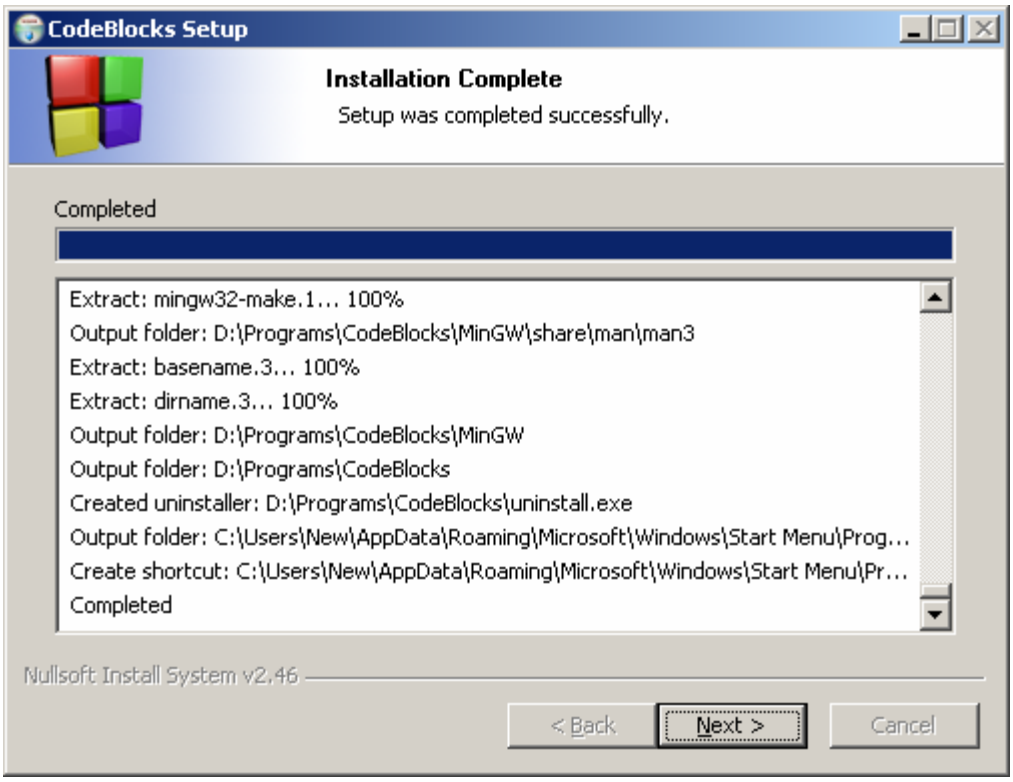

Рис. 21 – Программа установки (шаг 9)

Рис. 22 – Программа установки (шаг 10)

Ярлык для запуска Code::Blocks должен появиться на панели быстрого запуска. Также аналогичный ярлык должен появиться на рабочем столе.

Code::Blocks можно запускать с помощью этих ярлыков, а также с помощью стандартного системного меню программ.

При первом запуске Code::Blocks должно появиться окно автоматического нахождения компилятора (рис. 23).

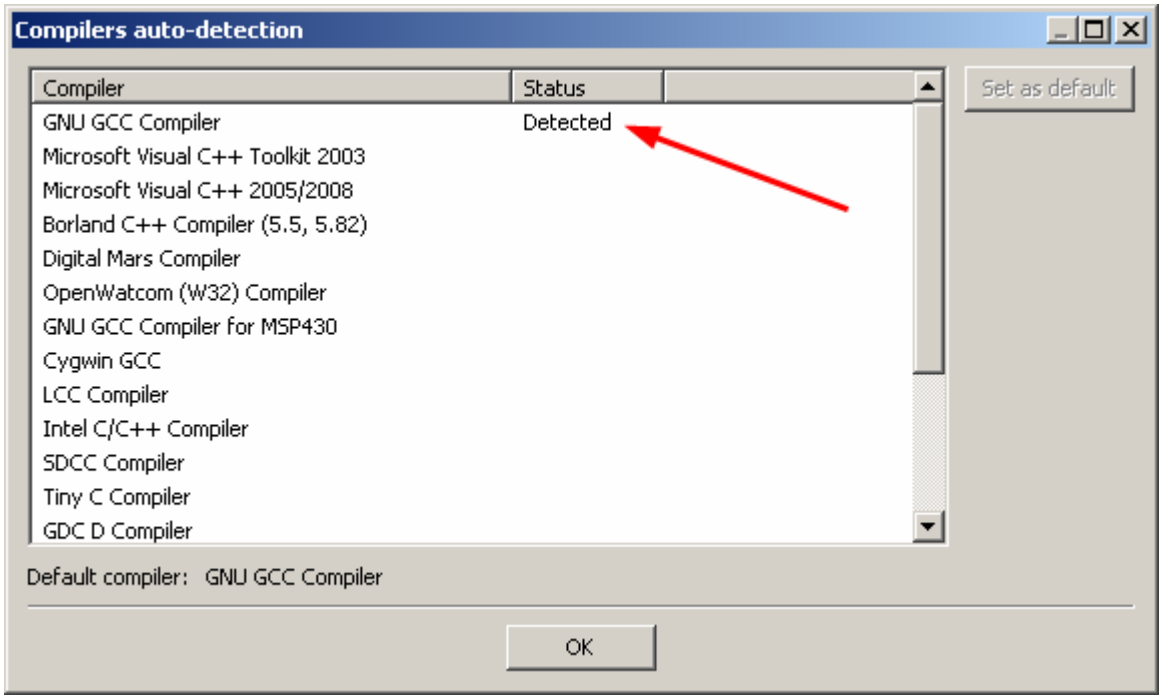

Рис. 23 – Окно автоматического нахождения компилятора

Заметьте, что в первой строке должно быть слово "Detected". Это означает, что найден компилятор. Надо нажать "OK". Процесс первого запуска продолжится, и будет запущена среда разработки. При этом появится окно с советом дня (рис. 24).

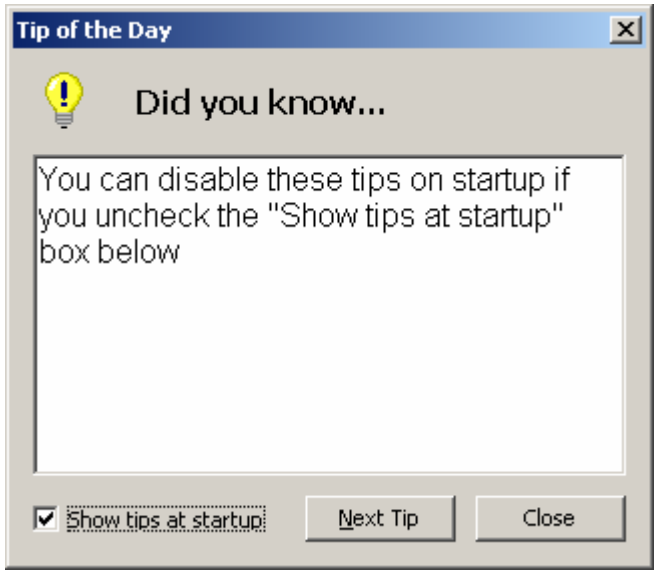

Рис. 24 – Окно с советом дня

Если вы не хотите, чтобы это окно появлялось при каждом новом запуске Code::Blocks, то тогда уберите галочку "Show tips at startup в этом окне". Нажмите "Close", чтобы закрыть это окно.

После этого может появиться следующее окно (рис. 25).

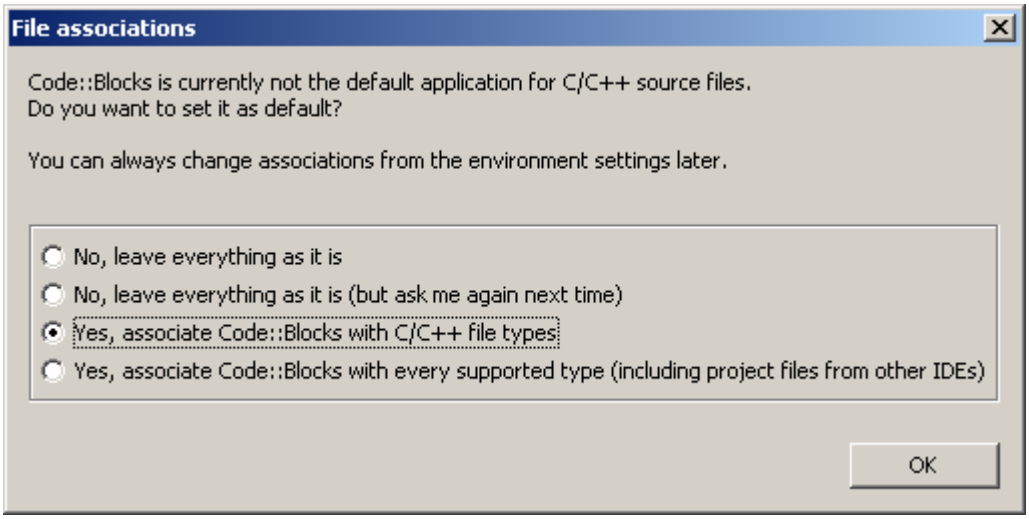

Рис. 25 – Окно установки связывания по расширению

Здесь лучше нажать "OK", чтобы файлы, относящиеся к C/C++, были связаны с Code::Blocks.

Также можно закрыть окно "Scripting console" (рис. 26).

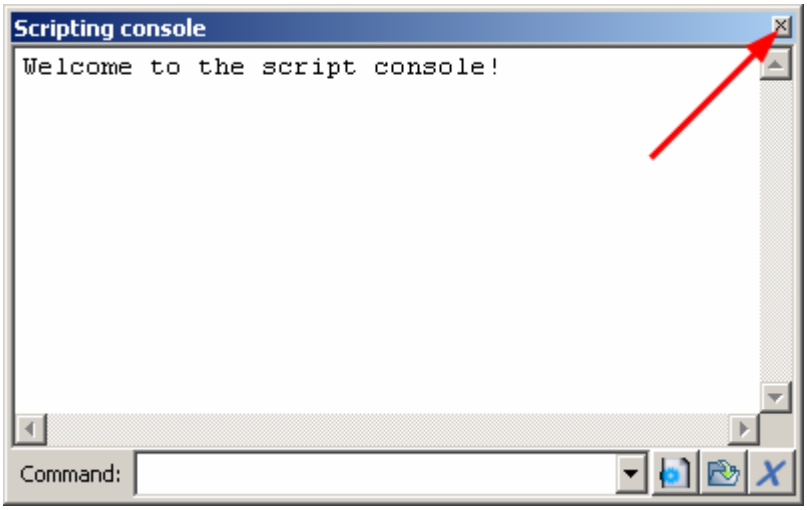

Рис. 26 – Окно «Scripting console»

Теперь процесс первого запуска можно считать законченным, и на экране будет программная среда Code::Blocks (рис. 27).

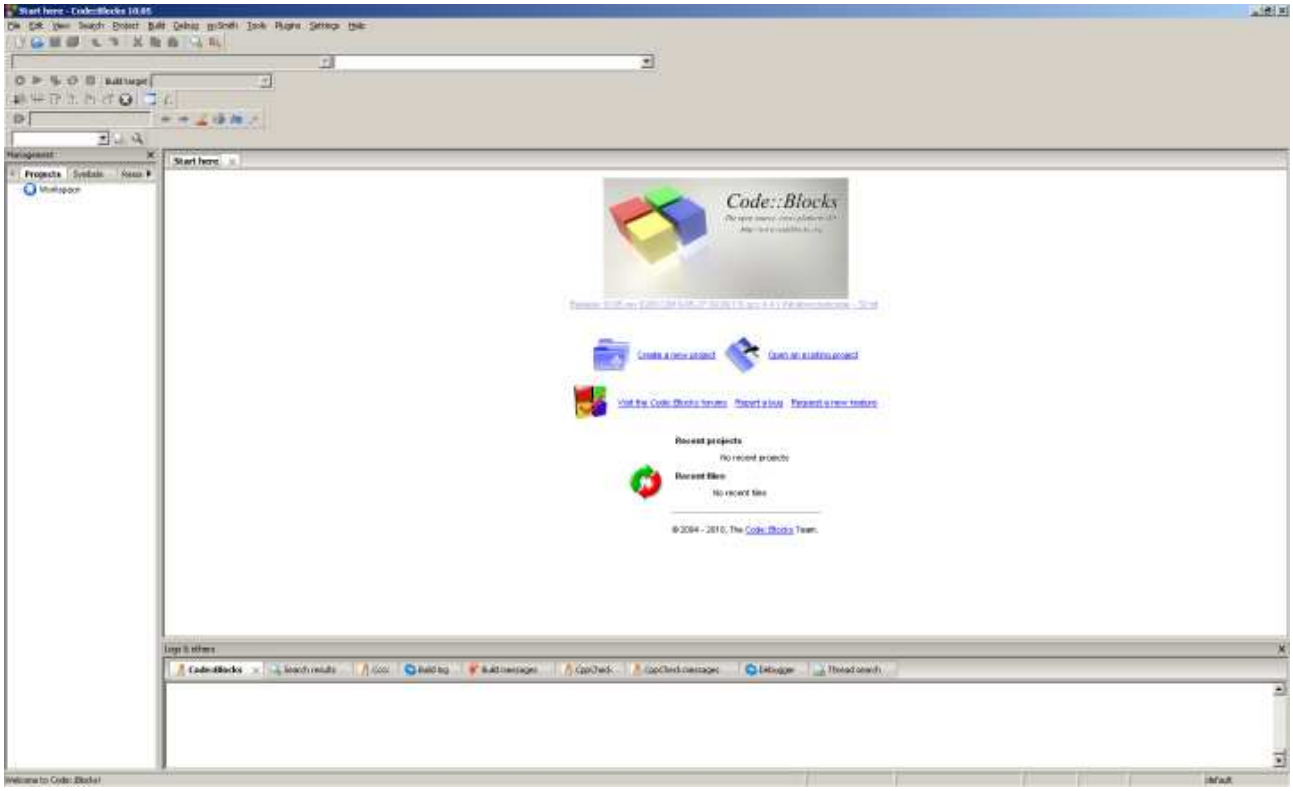

Рис. 27 – Программная среда Code::Blocks (начальный вид)

2.1.3.2. Настройка среды разработки

Для начала можно более компактно распределить панели инструментов в верхней части окна (рис. 28).

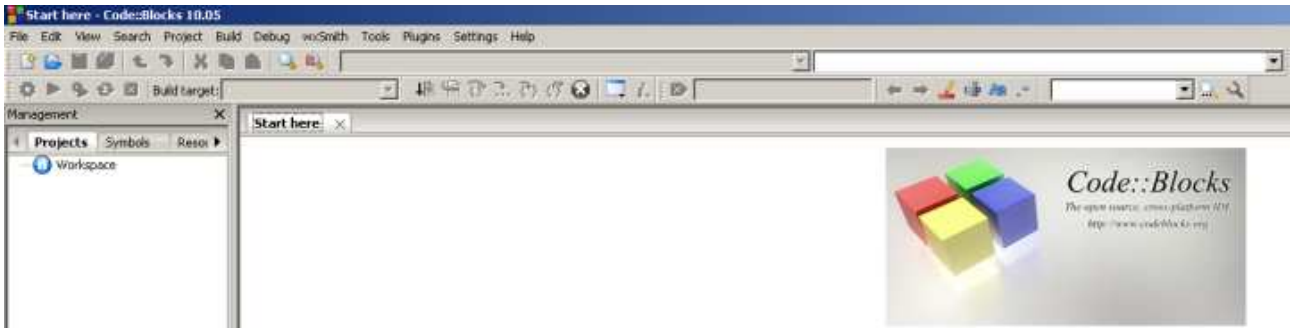

Рис. 28 – Расположение панелей инструментов в верхней части окна

После этого надо сделать несколько изменений в базовых настройках.

Выберите меню "Settings=>Compiler and debugger…". Появится окно настроек (рис. 29).

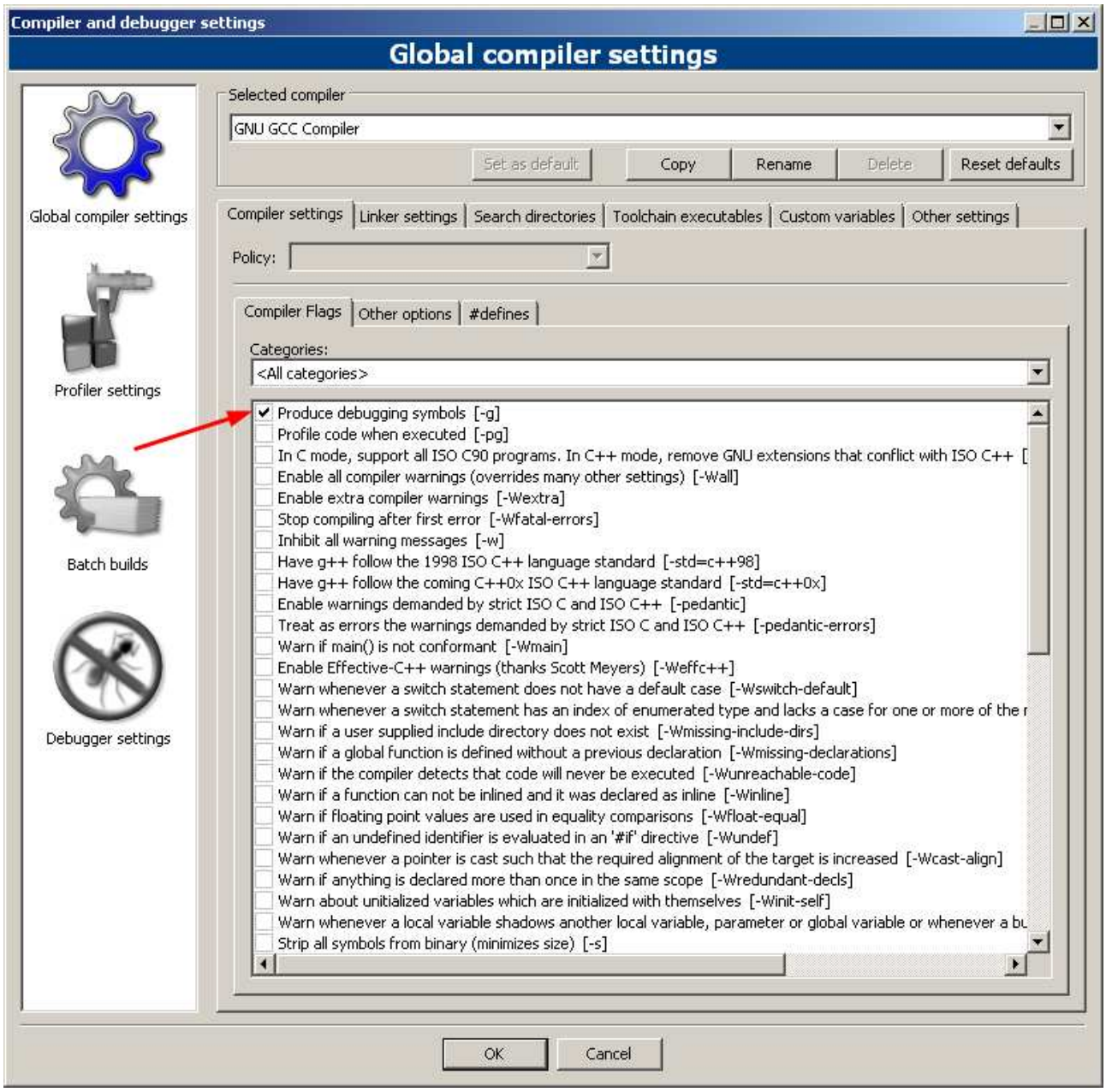

Рис. 29 – Настройка «Produce debugging symbols»

В этом окне надо поставить галочку для "Produce debugging symbols". Это изменение позволит выполнять режим отладки, который будет рассмотрен в ходе одной из последующих лекций.

Не закрывая это окно, надо нажать вкладку "Other options" (рис. 30).

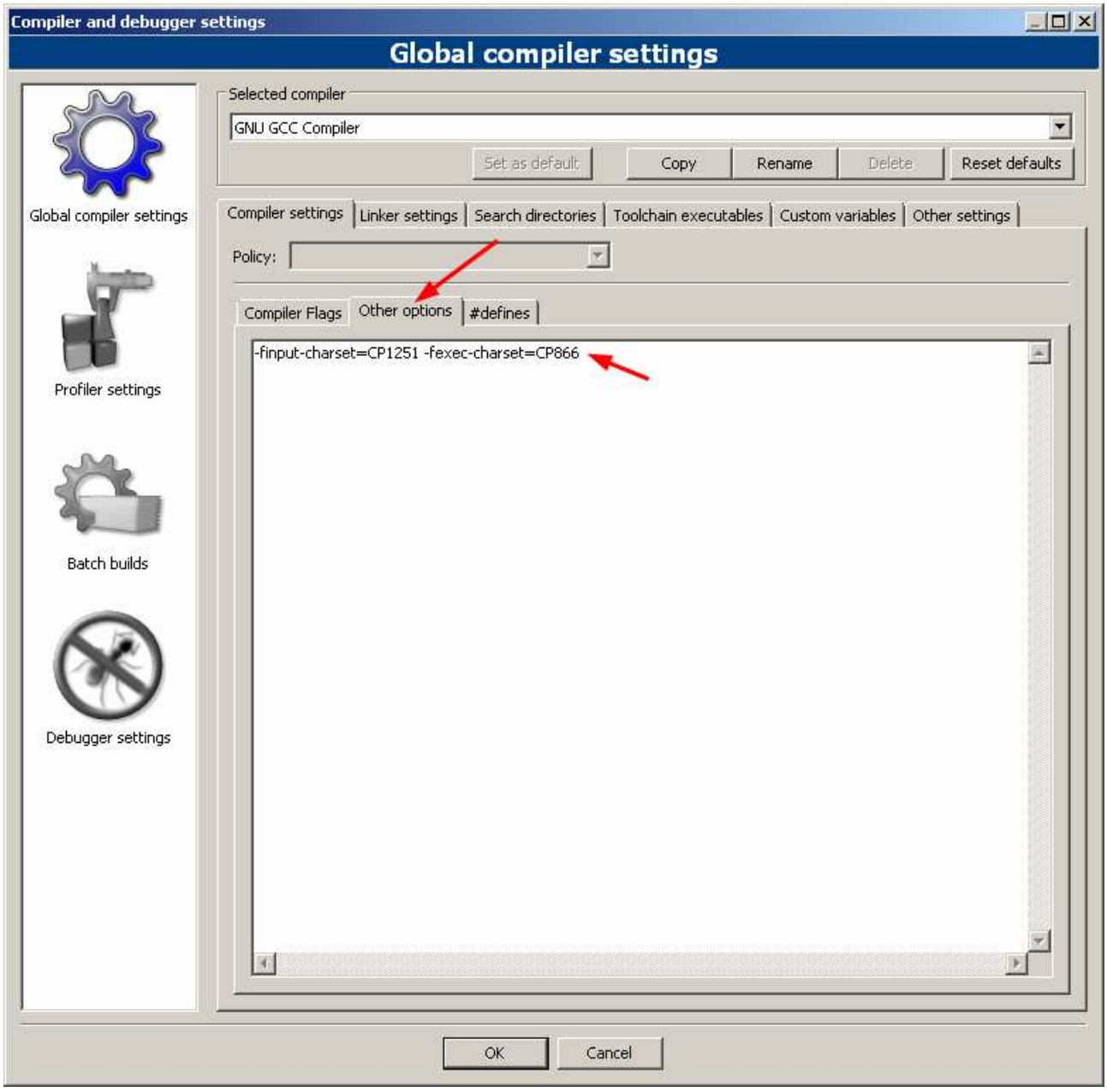

Рис. 30 – Настройка «Other options»

И в поле редактирования надо добавить следующую строку.

```
-finput-charset=CP1251 -fexec-charset=CP866
```

Это позволить правильно отображать кириллицу в консоли в случае операционной системы семейства Windows. При использовании Code::Blocks в других операционных системах (например, в Linux) эту строку добавлять не надо.

Не закрывая это окно, надо нажать слева на рисунок с надписью "Debugger settings", а затем поставить галочку для "Evaluate expression under cursor" (рис. 31).

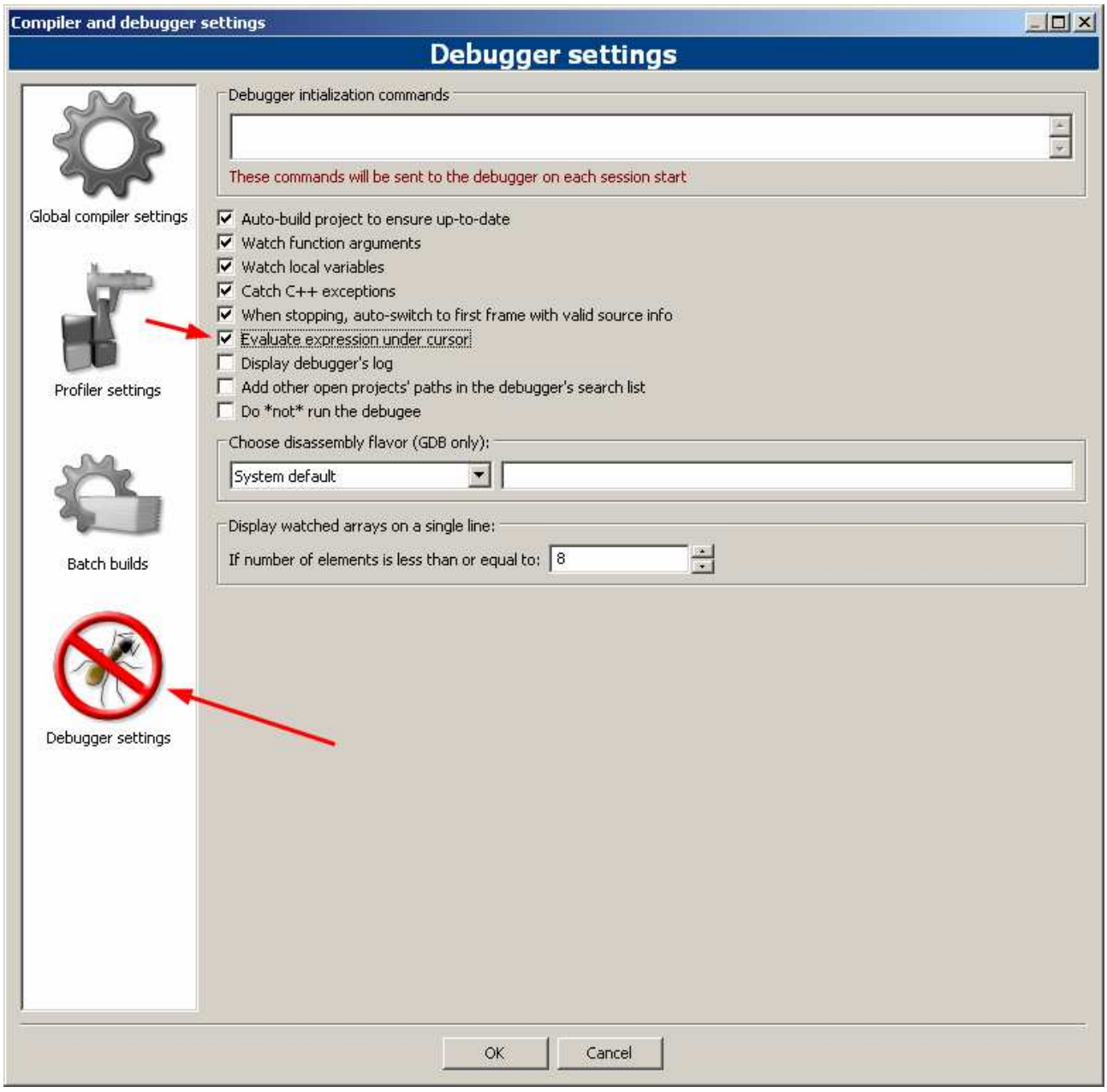

Рис. 31 – Настройка «Debugger settings»

После этого нажмите "OK". Все сделанные изменения должны вступить в силу.

2.1.3.3. Первый проект

Теперь попробуем создать программу на языке Си.

Для этого надо создать проект. Вообще, проект нужен для объединения в единое целое нескольких файлов. Но и для одного файла в данном случае удобнее сделать проект. Кроме того, в дальнейшем тогда будет привычнее работать именно с проектом.

Итак, для создания проекта выберите меню "File", затем в появившемся списке выберите "New", затем в появившемся списке выберите "Project" (для краткости подобное действие можно обозначить как "File=>New=>Project", и в дальнейшем будут использоваться такие более краткие обозначения). В итоге, должно появиться диалоговое окно (рис. 32).

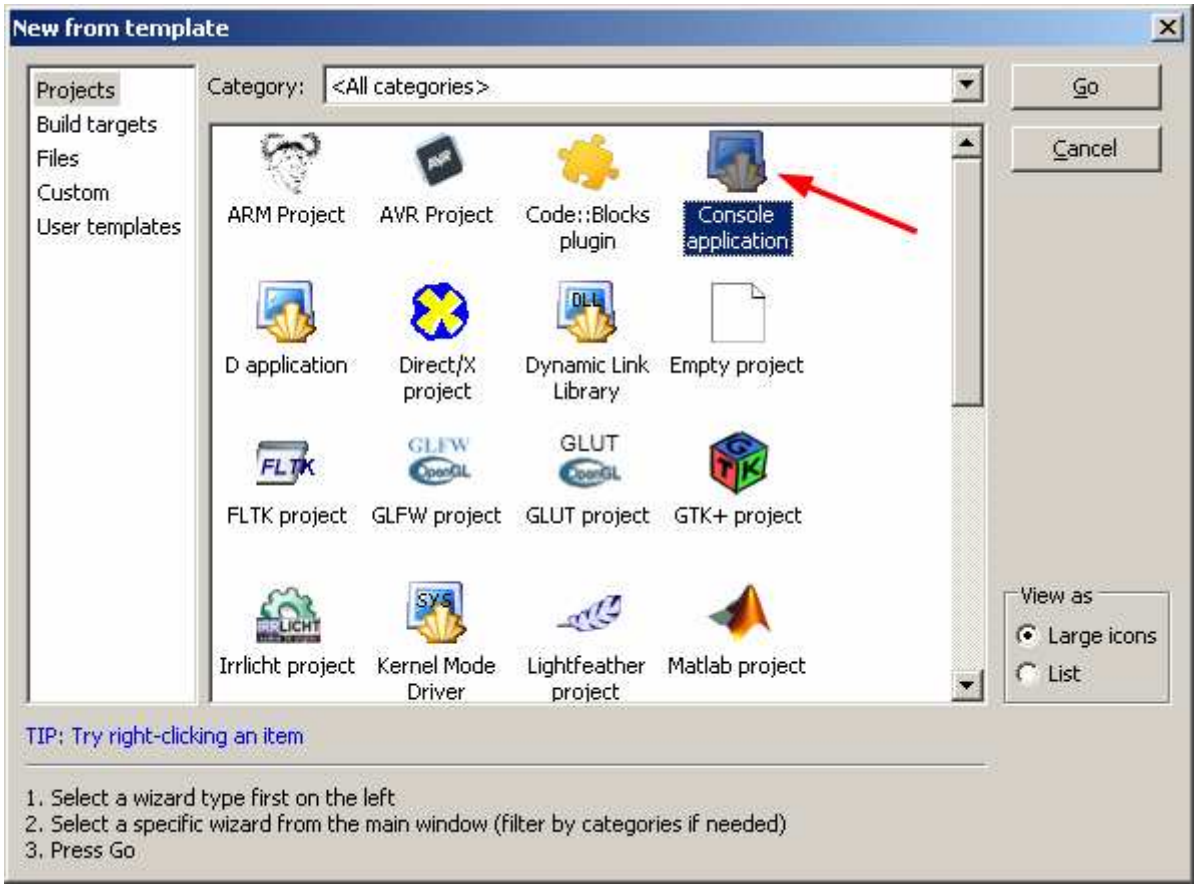

Рис. 32 – Окно выбора вида проекта

В нем выберите "Console application" и нажмите "Go". Появится мастер создания проекта (рис. 33).

Рис. 33 – Мастер создания проекта (шаг 1)

Нажмите "Next". Появится следующее окно (рис. 34).

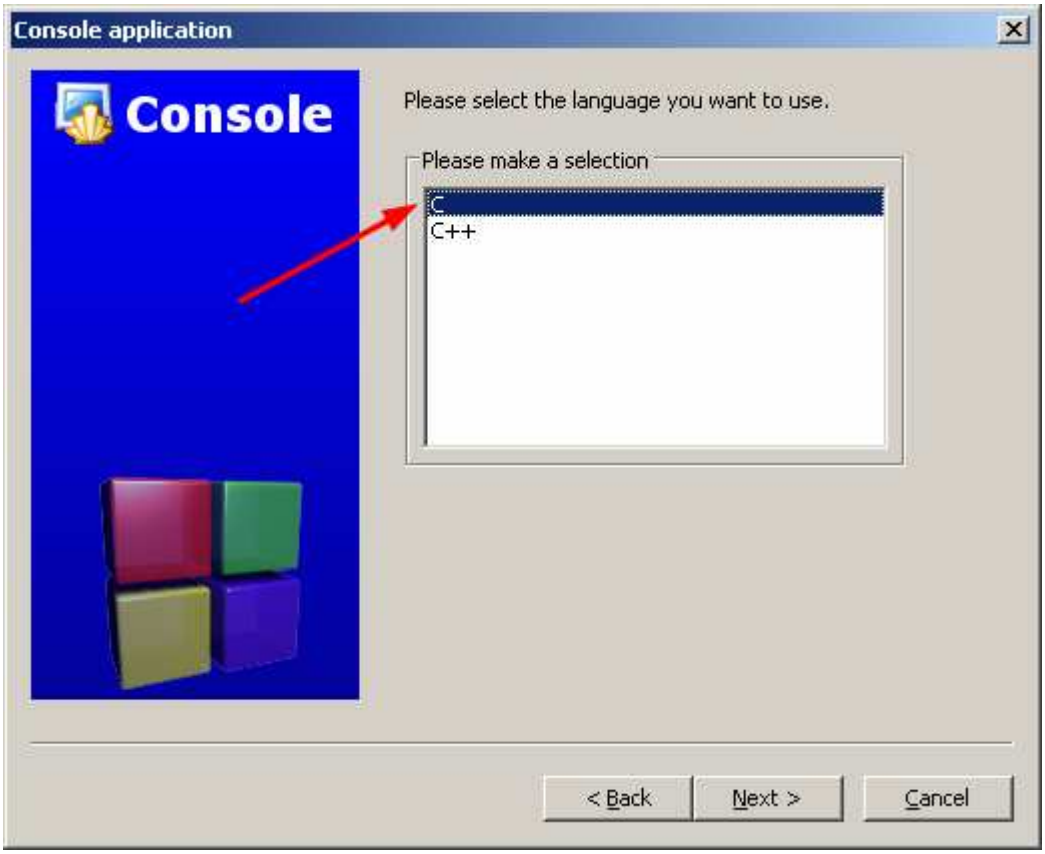

Рис. 34 – Мастер создания проекта (шаг 2)

Здесь надо выбрать C (а не C++) и нажать "Next". Появится следующее окно (рис. 35).

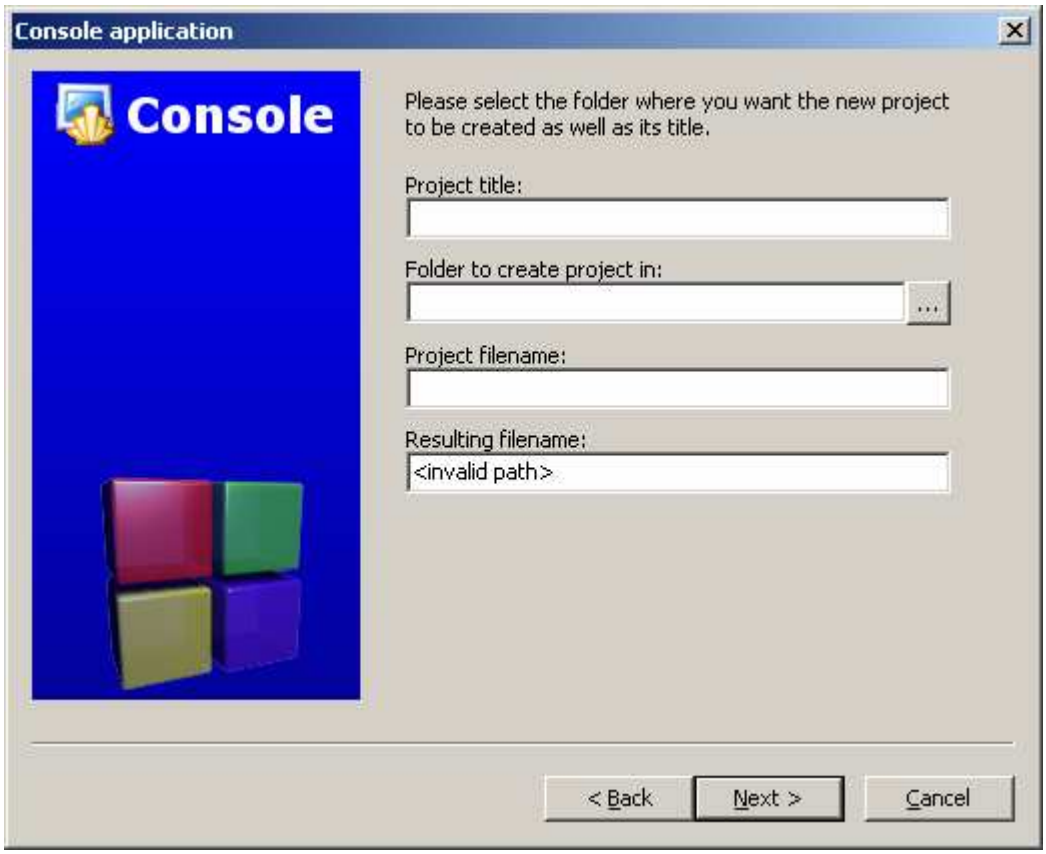

Рис. 35 – Мастер создания проекта (шаг 3)

В строке "Project title" введите название проекта, например, Project01. Заметьте, что в строке "Project filename" (имя файла проекта) это название будет скопировано. Это имя файла можно изменить (чтобы оно отличалось от названия проекта), но для удобства лучше оставить их одинаковыми. Затем, в строке строки "Folder to create project in" (папка для создания в ней проекта) надо указать папку, где будет располагаться папка с проектом. Можно либо выбрать какую-нибудь папку из имеющихся с помощью кнопки "…", либо вручную набрать имя папки, например, C:\Projects.

Тогда именно в выбранной папке будет создана папка с названием проекта, где и будет расположен сам проект. Можете это проверить, обратив внимание на строку "Resulting filename" (рис. 36).

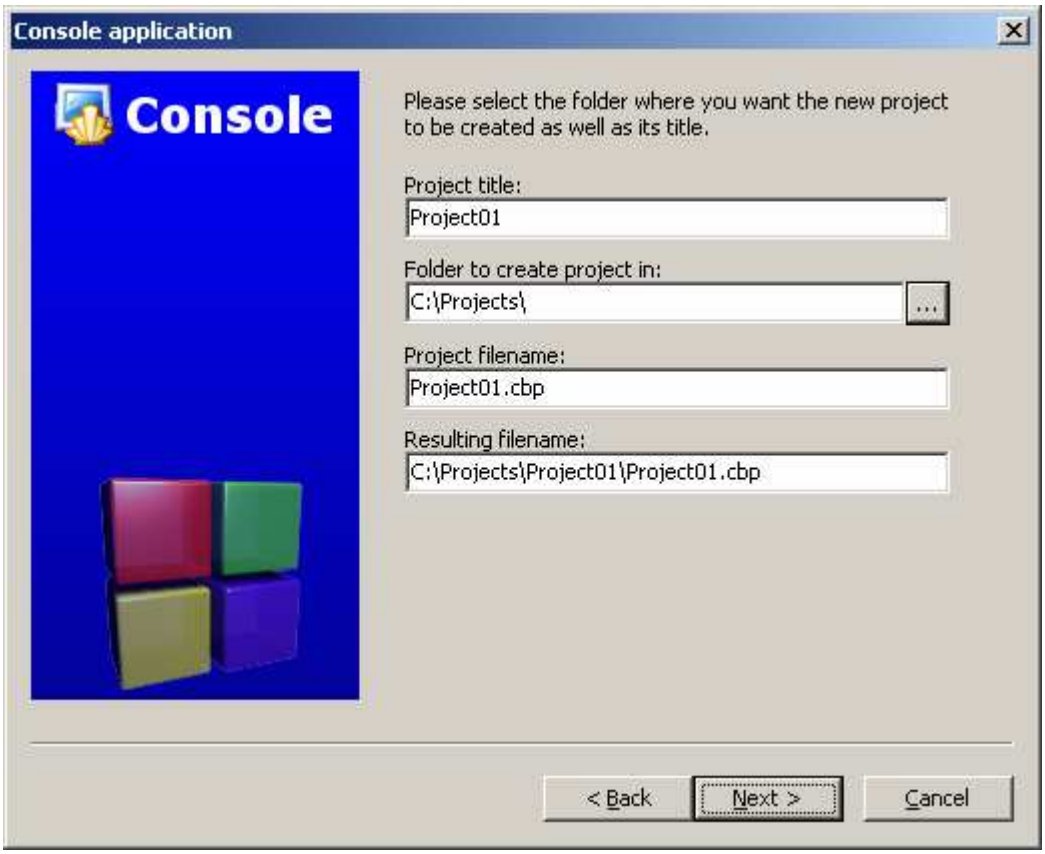

Рис. 36 – Мастер создания проекта (шаг 4)

При задании имени проекта и папки для проекта не допускайте использование имен папок с пробелами или буквами, которые не являются латинскими (строка "Resulting filename" не должна содержать пробелов в именах папок, а также не должна содержать других букв, кроме латинских).

Нажмите Next. Появится следующее окно (рис. 37).

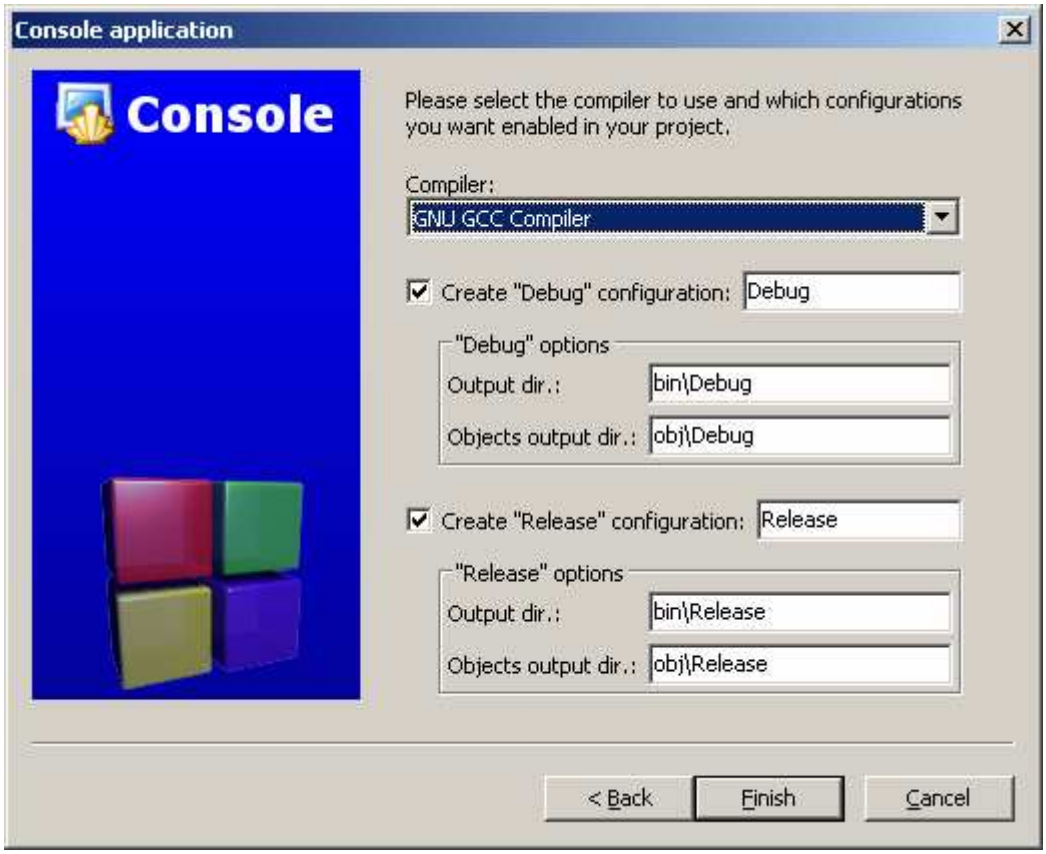

Рис. 37 – Мастер создания проекта (шаг 5)

Эти настройки оставьте без изменений. Нажмите "Finish". Проект будет создан.

В левой верхней части (в окне "Management") отобразится проект с папкой "Sources" (рис. 38).

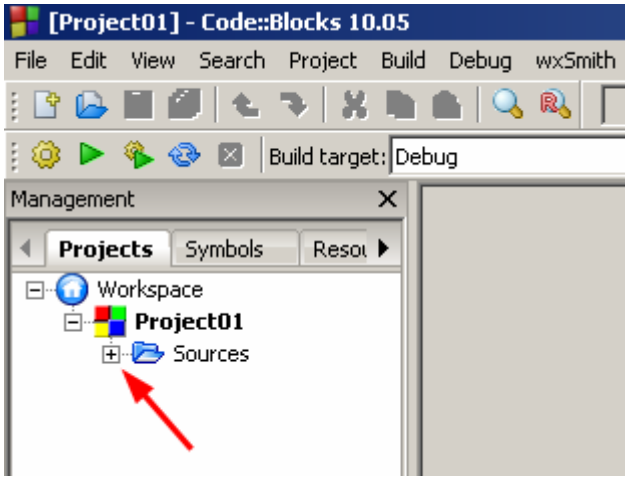

Рис. 38 – Отображение проекта с папкой «Sources»

Надо нажать на "+" (или дважды нажать на папку) и эта папка откроется (рис. 39).

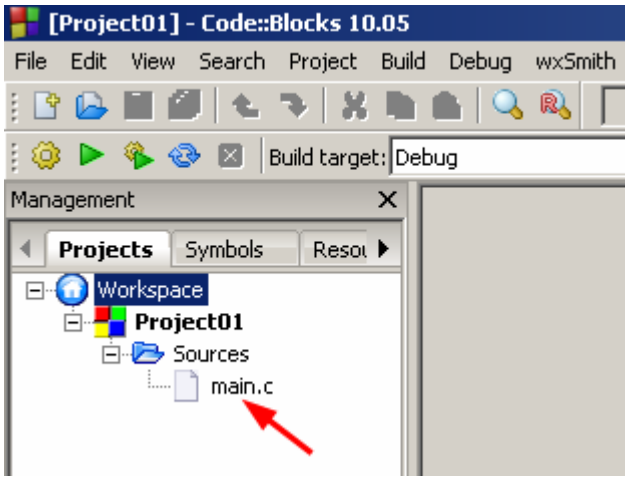

Рис. 39 – Отображение проекта с открытой папкой «Sources»

В ней будет единственный файл main.c. Надо дважды нажать на него, тогда справа откроется поле редактирование, в котором отобразится содержимое этого файла (в верхней части будет закладка с именем файла) (рис. 40).

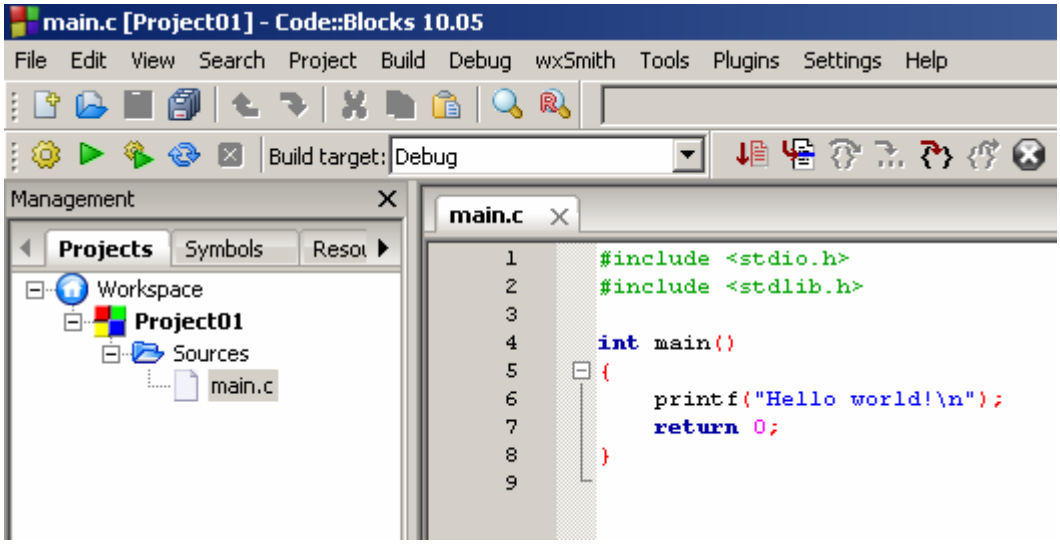

Рис. 40 – Поле редактирования с автоматически созданной программой

В окне редактирования будет текст первой программы с подсветкой. Обратите внимание, что для разных частей текста используются свои цвета, что облегчает чтение текста.

Это первая простейшая программа, которая выводит текст "Hello world!" ("Привет, мир!"). Традиционно, именно такая программа используется в качестве первой программы при знакомстве с языком программирования. В случае Code::Blocks эта программа автоматически создается при создании нового проекта.

Размер шрифта в окне редактирования можно очень просто изменить. Для этого, вращайте колесо на мыши, удерживая нажатой клавишу Ctrl.

Еще можно использовать меню "Settings=>Editor" и, нажав в верхней правой части кнопку "Choose", поменять не только размер шрифта, но и сам шрифт. Только учтите, что для написания программ лучше использовать шрифты, у которых все буквы имеют одинаковую ширину. К таким шрифтам, к примеру, относятся шрифты: Courier New, Courier, Liberation Mono, Lucida Console, Monospace и др.

Если окажется, что на экране отсутствует окно "Management" или какое-нибудь другое нужное окно, то тогда надо выбрать меню View и в поя-

вившемся меню выбрать пункт, соответствующий нужному окну.

Если вдруг проект оказался закрытым, например, при выходе и повторном заходе в Code::Blocks, то его можно заново открыть. Для этого надо выбрать меню "File=>Recent projects", где, затем, в появившемся списке следует выбрать нужный проект. Либо можно использовать меню "File=>Open", после чего выбрать файл Project01.cbp.

2.1.3.4. Сборка и запуск программы

Эта программа будет детально рассмотрена чуть позже, а сейчас попробуем ее запустить.

Для этого должна быть выполнена компиляция текста программы (Compiling), и с помощью компоновки (Linking) должен быть создан исполняемый файл с расширением .exe, который и будет запускаться. Весь этот процесс компиляции и компоновки называется сборкой (Building).

Надо отметить, что процесс, обозначаемый здесь словом Compiling, также часто называют процессом трансляции. Есть разные варианты терминологии в этой области. Например, указанный выше процесс сборки может называться компиляцией, которая, свою очередь, состоит из этапов трансляции и компоновки. Но мы сейчас не будет углубляться в вопросы терминологии, и просто в качестве основы будем использовать англоязычный вариант терминов, естественно, с переводом на русский язык. Таким образом, мы будет говорить о сборке (Building), состоящей из этапов компиляции (Compiling) и компоновки (Linking). Этот вариант в данном случае кажется более удобным, так как в соответствующие наименования на английском языке можно наблюдать в процесса сборки.

Интегрированная среда разработки Code::Blocks позволяет автоматизировать сборку и запуск (Run) программы. Для сборки и запуска программы достаточно выполнить команду "Собрать и запустить" (Build and run), нажав кнопку  или клавишу F9. Еще один вариант – это выбрать

меню "Build=>Build and run".

В нижнем окне (сообщений о процессе сборки) будут появляться надписи "Compiling", "Linking" и т.д., что отражает ход компилирования и компоновки программы (рис. 41).

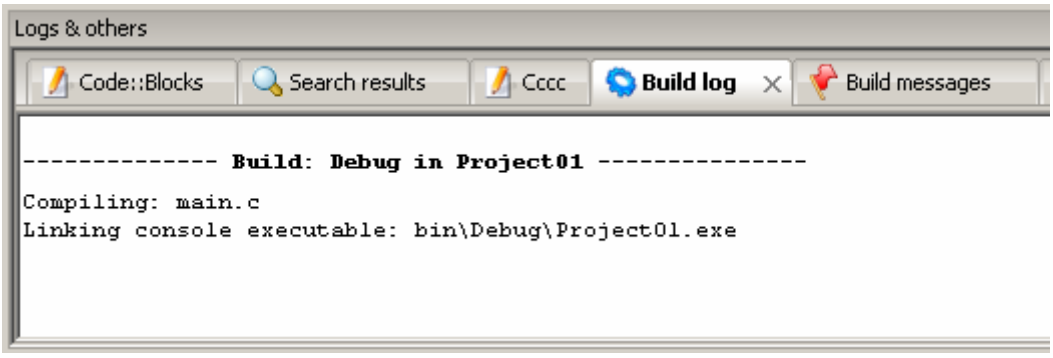

Рис. 41 – Пример сообщений о процессе сборки

В результате должно появиться консольное окно, где в верхней части будет выведено предложение, указанное в программе в кавычках, а именно предложение "Hello world!" (рис. 42).

Рис. 42 – Консольное окно с текстом, формируемом при запуске программы

Таким образом, программа выводит на экран заданную строку.

Под этой строкой будет две строки. Первая из них выдает код возврата программы и время, затраченное на выполнение программа. Вторая выдает сообщение о том, что для продолжения надо нажать любую клавишу.

Эти две строки вырабатываются автоматически, если программа запускается из Code::Blocks.

Шрифт, используемый в консольном окне при необходимости можно поменять. Например, в случае операционной системы семейства Windows, для этого можно попробовать выполнить следующие действия.

Нажмите правой кнопкой мыши на значок в верхнем левом углу консольного окна. Должно появиться всплывающее меню, где надо выбрать пункт "Умолчания".

Появится окно настроек (рис. 43).

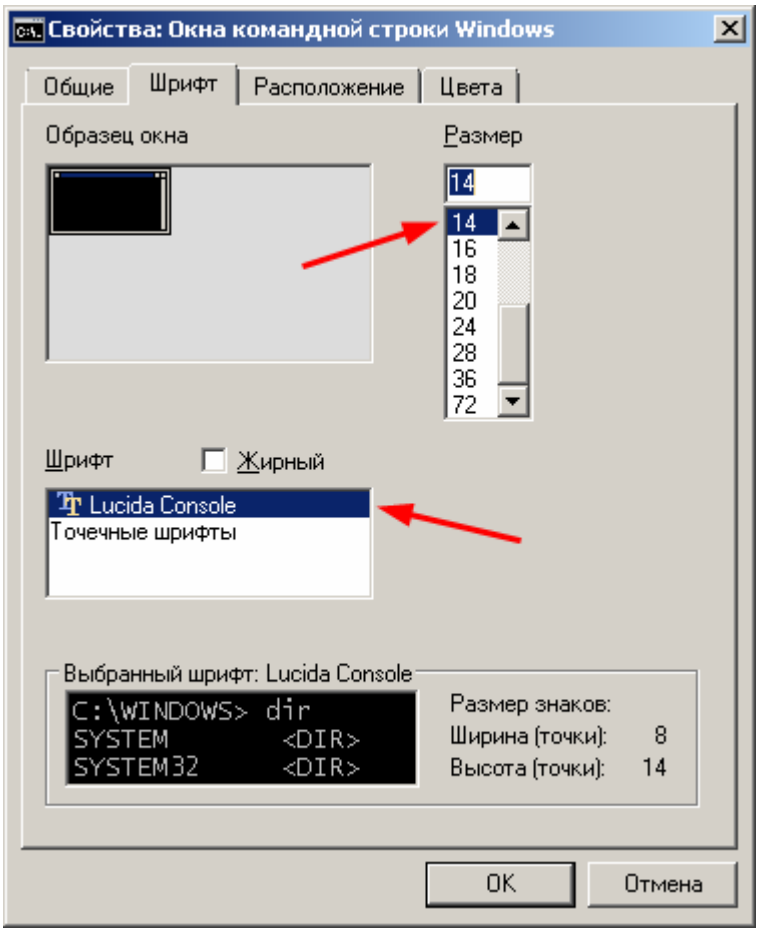

Рис. 43 – Настройка шрифта для консольного окна

В этом окне надо выбрать нужный шрифт (например, Lucida Console). Затем, для этого шрифта надо выбрать желаемый размер (например, 14). После этого надо нажать "OK", чтобы сохранить настройки.

Затем, надо закрыть консольное окно, нажав любую клавишу.

После этого надо заново запустить программу, нажав кнопку  или клавишу F9 (или выбрав меню "Build=>Build and run").

Теперь консольное окно должно иметь новый шрифт.

2.1.3.5. Сообщения о предупреждениях и ошибках в программе

При сборке программы (при попытке ее запуска после изменений) может происходить вывод предупреждений и ошибок. Как это происходит, рассмотрим на примере. Попробуйте запустить следующую программу.

```
#include <stdio.h>
int main()
{
   printf("Hello world!!!\n");
}
```

Эта программа не содержит return, но она будет запущена. Попробуйте ее запустить.

Однако в процессе компиляции будет выдано предупреждение, связанное с тем, что нет оператора return. При запуске программы это предупреждение исчезает, так как оно не влияет на запуск программы. Но его можно прочитать, если выполнить только построение файла (без запуска приложения). Для этого надо сформировать команду "Собрать" (Build), нажав Ctrl-F9, или нажав кнопку  , или выбрав меню "Build=>Build".

Но если с предыдущей сборки или запуска программа не менялась, то новая сборка выполняться не будет. И в окне внизу будет выведено сообщение о том, что сборка уже самая свежая, и больше ничего не надо делать для этой команды ("Target is up to date. Nothing to be done.") (рис. 44).

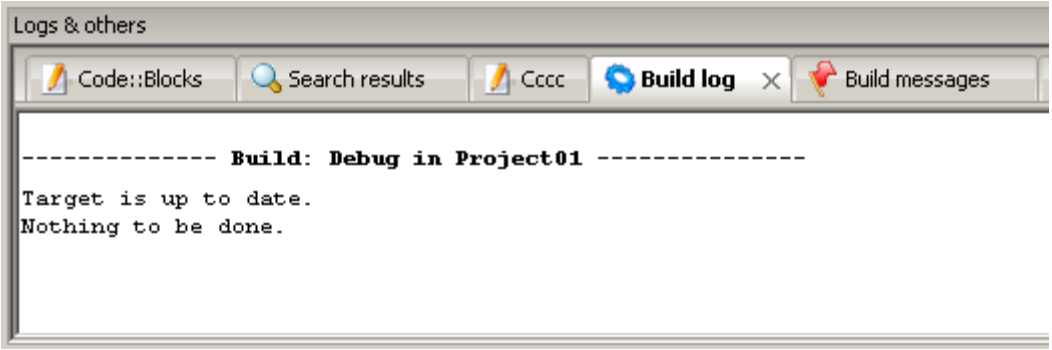

Рис. 44 – Сообщение о том, что сборка уже является самой свежей

В таком случае надо собрать заново. Для этого надо сформировать команду "Собрать заново" (Rebuild), нажав Ctrl-F11, или нажав кнопку , или выбрав меню "Build=>Rebuild". После формирования этой команды появляется окно для ее подтверждения. Чтобы подтвердить эту команду надо нажать "Yes" в указанном окне.

Тогда в нижнем окне можно увидеть строку с предупреждением (синего цвета), а также сообщение о наличии одного предупреждения (1 warnings) (рис. 45).

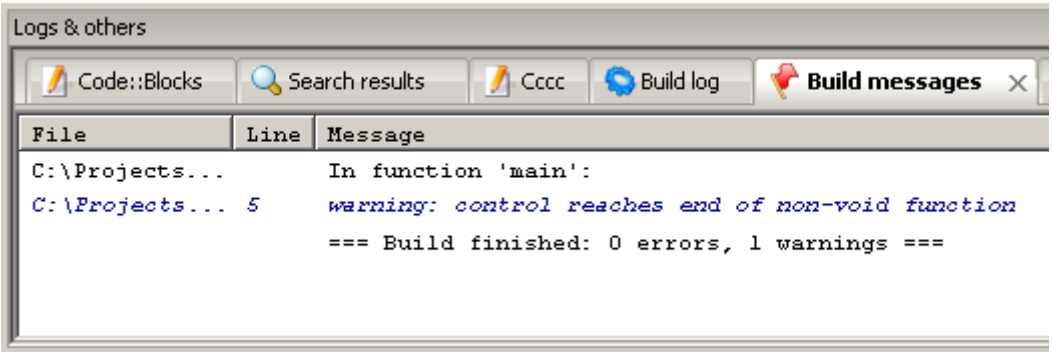

Рис. 45 – Пример вывода предупреждения

Вообще, предупреждение позволяет выполнить программу, но сигнализирует программисту о возможных ошибках или неточностях.

В дальнейшем оператор return для функции main можно не использовать для краткости, как это сделано в . Но надо учитывать, что из-за этого появляется предупреждение.

Теперь попробуйте запустить следующую программу.

```
#include <stdio.h>
int main()
{
  printf(Hello world!!!\n);
}
```

Заметьте, что программа не будет запущена. Вместо этого в нижнем окне появятся строки красным шрифтом, которые сообщают об ошибках (рис. 46).

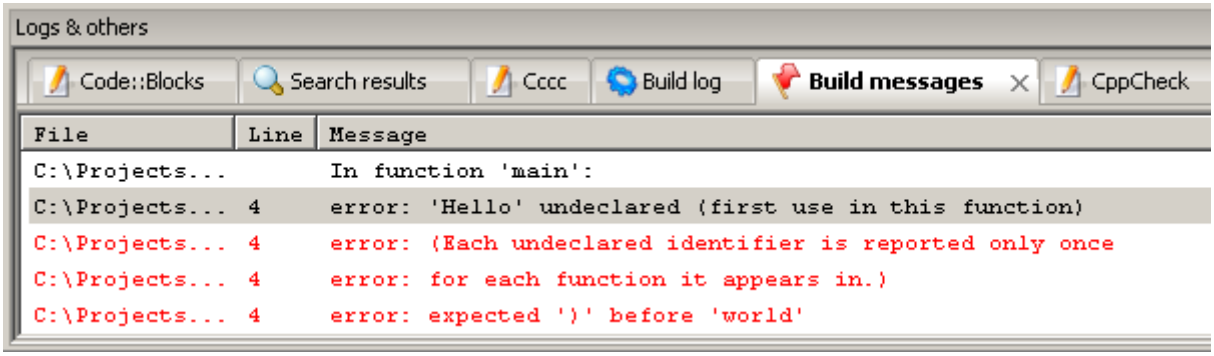

Рис. 46 – Пример вывода сообщений об ошибках

Строка в программе, где имеется ошибка, подсвечивается красным квадратом (рис. 47).

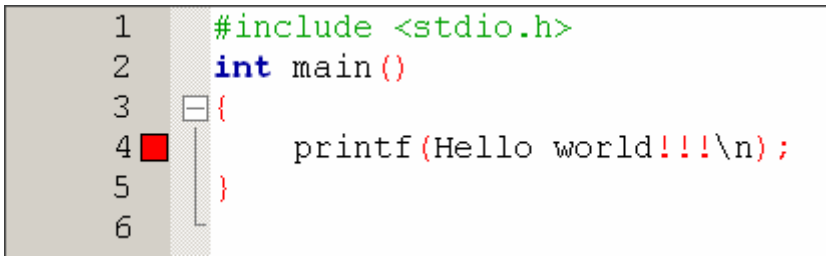

Рис. 47 – Пример индикации ошибки в тексте программы

Надо учитывать, что одна действительная ошибка может приводить сразу к нескольким сообщениям об ошибке. В данном примере формируется 5 ошибок, поэтому выводится 5 строк красным шрифтом.

Анализируя сообщения об ошибках и текст программы, нетрудно понять, что ошибка возникает из-за того, что в строке «*printf* (Hello world!!!\n);» не указаны кавычки. Соответственно выводится ошибка

"'Hello' undeclared". Слово "undeclared" означает "не объявлено", то есть компилятор пытается понять, что обозначает Hello, и что при этом надо делать. Он не может найти это слово среди известных слов, поэтому выдает ошибку. Если же это слово находится в двойных кавычках, то тогда оно воспринимается как некоторая строка, внутри которой может быть что угодно. Строку можно выводить на экран. Поэтому после исправления программы (добавления кавычек, где надо) все должно быть нормально.

Исправьте программу и проверьте ее правильность выполнения.

Нетрудно заметить, что ошибка может содержаться рядом со строкой, отмеченной красным квадратом. Поэтому при возникновении ошибки надо анализировать также и соседние строки.

При выходе из Code::Blocks могут появляться окна с кнопками «Да», «Нет» (или «Yes», «No»). Там, как правило, спрашивается о необходимости сохранения измененного файла (или проекта), либо измененных настроек среды разработки. Нажимайте «Да» («Yes»), если желаете эти изменения сохранить.

Таким образом, мы выполнили базовое знакомство с интегрированной средой разработки Code::Blocks.

2.1.3.6. Средства отладки

Сейчас разберем, как происходит отладка программы. В частности, рассмотрим пошаговое выполнение. Этот процесс помогает:

1) найти ошибки при выполнении программы (особенно, когда сообщения об ошибках не выводятся, а программ дает неправильный результат);

2) понять процесс выполнения программы.

В качестве примера программы возьмите следующую программу.

```
#include <stdio.h>
main()
```

```
{
  int x=0; double y=10.0; /* x=0 y=10.0 */
  printf("x=%d y=%.1f \n",x,y);
  x=y*2; /* x=20 y=10.0 */
  printf("x=%d y=%.1f \n",x,y);
  x++;
  y-=5; /* x=21 y=5.0 */
  printf("x=%d y=%.1f \n",x,y);
}
```

Эта программа изменяет значения переменных x, y, а также выводит на экран три строки, отображающие текущее состояние переменных. В комментариях указывается состояние переменных после выполнения соответствующей строки.

Запустите программу и убедитесь, что все три строки выводятся одновременно, и они соответствуют значениям переменных, которые указаны в комментариях.

Но процесс изменения x, y можно сделать пошагово, чтобы проследить, как меняются их значения. При этом строки будут выводится последовательно, шаг за шагом.

Для этого закройте консольное окно (если оно открыто). Поставьте курсор на строку, следующую сразу после открывающей фигурной скобки, то есть на строку, содержащую int x=0 (рис. 48).

main.c ×

```
1  #include <stdio.h>
2  main()
3  {
4   int x=0; double y=10.0; /* x=0 y=10.0 */
5   printf("x=%d y=%.1f \n",x,y);
6   x=y*2; /* x=20 y=10.0 */
7   printf("x=%d y=%.1f \n",x,y);
8   x++;
9   y-=5; /* x=21 y=5.0 */
10  printf("x=%d y=%.1f \n",x,y);
11 }
12
```

Рис. 48 – Установка курсора перед началом режима отладки

После этого нажмите кнопку  (или выберите меню Debug–>Run to cursor, или нажмите клавишу F4). Это соответствует команде «Запустить до курсора» (Run to cursor). Начнется процесс отладки, в нижнем окне будут появляться строки сообщений. Дождитесь, пока напротив курсора не появится желтый треугольник (рис. 49).

main.c ×

```
1  #include <stdio.h>
2  main()
3  {
4   int x=0; double y=10.0; /* x=0 y=10.0 */
5   printf("x=%d y=%.1f \n",x,y);
6   x=y*2; /* x=20 y=10.0 */
7   printf("x=%d y=%.1f \n",x,y);
8   x++;
9   y-=5; /* x=21 y=5.0 */
10  printf("x=%d y=%.1f \n",x,y);
11 }
12
```

Рис. 49 – Индикация текущий позиции отладки

Этот треугольник указывает, на каком месте было приостановлено выполнение программы. Сейчас программа ожидает продолжения выполнения с этого места.

Для выполнения строки, на которую указывает треугольник, надо нажать  (или выбрать меню Debug–>Next line, или нажать клавишу F7).

После выполнения указанного действия треугольник переместится на следующую строку. Таким образом, предыдущая строка выполнена, и переменные x и y должны принять значения, указанные в комментарии.

Но эти значения можно проверить. Действительно ли эти значения присвоены данным переменным?

Текущие значения переменных можно проверять в ходе отладки. Для этого есть два способа.

Первый способ состоит в том, что к нужной переменной подводится курсор мыши. Попробуйте сделать это для переменной y. При этом должно появиться всплывающее окно с информацией (рис. 50).

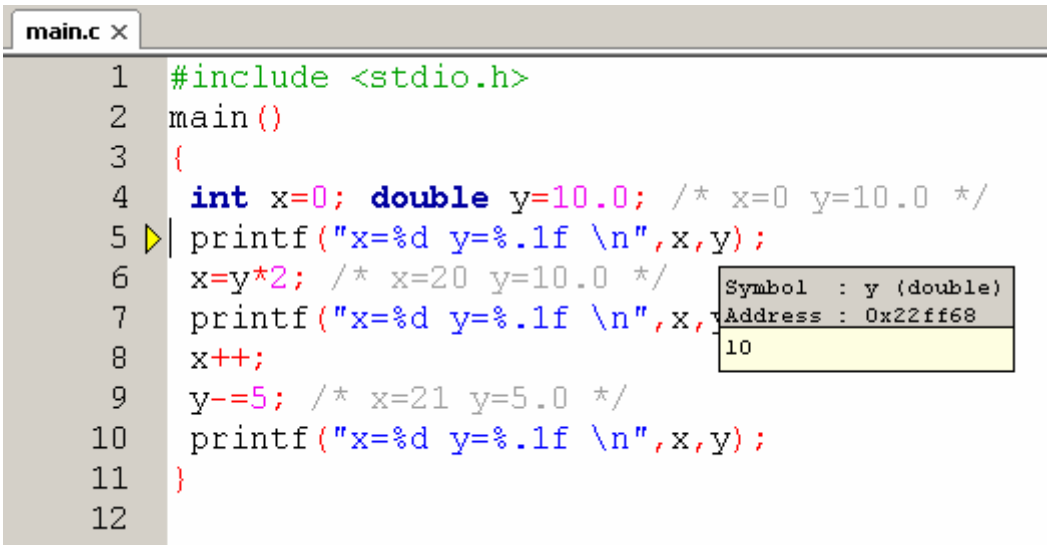

Рис. 50 – Пример всплывающего окна с информацией о переменной

В этом окне на желтом фоне отображается значение переменной. На сером фоне отображается дополнительная информация (имя, тип и адрес, по которому переменная расположена в памяти). Это окно вызывается, если подвести курсор к любому изображению переменной y в текстовом окне. Проверьте это. Аналогичные действия попробуйте сделать для переменной x.

Второй способ позволяет одновременно наблюдать значения многих переменных. Надо выбрать меню Debug–>Debugging windows–>Watches (рис. 51).

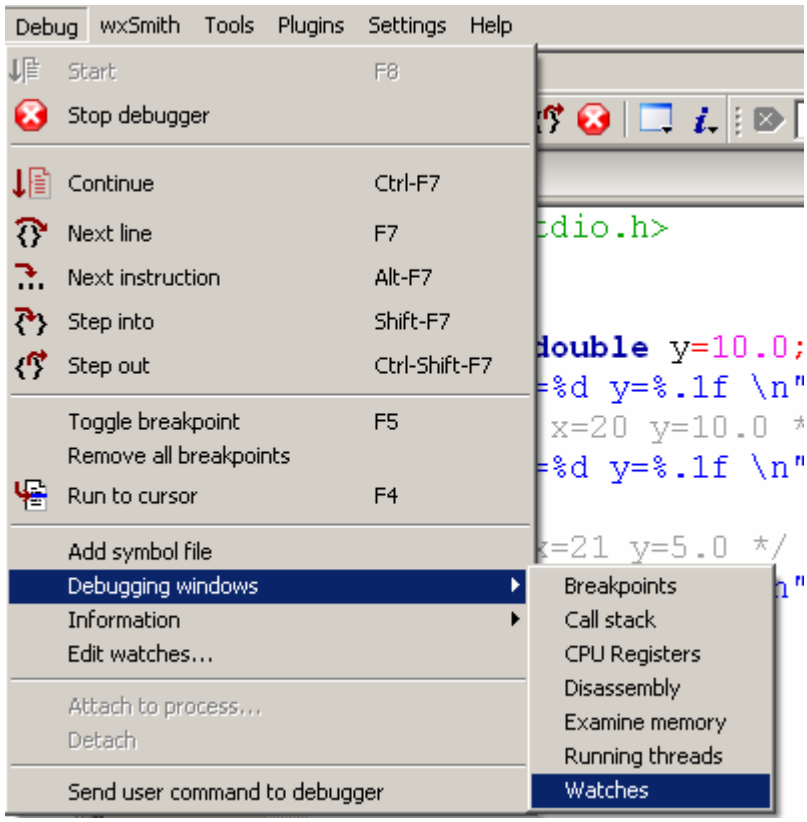

Рис. 51 – Активизация окна Watches

При этом должно появиться окно Watches (рис. 52).

Рис. 52 – Начальный вариант окна Watches

Раскройте Local variables. Вы увидите значения переменных (рис. 53).

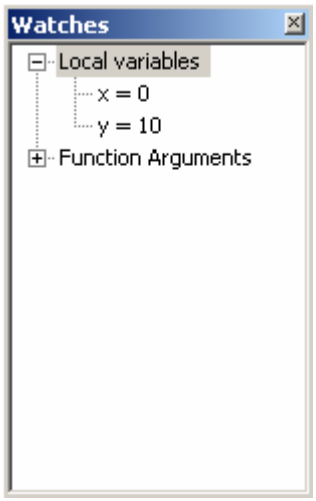

Рис. 53 – Раскрытие блока локальных переменных в окне Watches

Значения этих переменных соответствуют тем, что отображались во всплывающих окнах.

Окно Watches можно переместить в нужную часть экрана. Также можно поменять его размеры. Если окно переместить, например, в правый край экрана, то оно установится без перекрытия других окон. Аналогичным образом его можно переместить в другую область экрана.

Теперь попробуйте сделать следующий шаг отладки (переместить желтый треугольник). Для этого надо нажать  (или выбрать меню Debug–>Next line, или нажать клавишу F7).

Заметьте, что значения переменных не изменяться (см. окно Watches), так как в программе в этой строке не было сделано их изменения. Зато был произведен вывод строки в консольное окно. В этом можно убедиться, переключившись на консольное окно. Чтобы на него переключиться, можно воспользоваться клавишами Alt-Tab или просто выбрать его в панели задач.

Консольное окно будет иметь следующий вид (рис. 54).

Рис. 54 – Пример вывода текста в консольном окне в ходе отладки

Заметьте, что выведена только строка, соответствующая первому printf.

Теперь попробуйте самостоятельно перемещаться шаг за шагом по программе. При этом проверяйте значения переменных и состояние консольного окна.

Когда дойдете до конца программы, желтый треугольник исчезнет. После этого сделайте еще один шаг. Консольное окно должно закрыться, и выполнение программы должно завершиться.

Вот таким образом можно выполнять отладку программу.

Кратко рассмотрим другие возможности по отладке программы.

Существует команда «Переход внутрь» (Step into), которая вызывается нажатием  (или нажатием Shift-F7, или выбором меню Debug–>Step into). Эта команда позволяет «зайти» внутрь функции, которая вызывается в текущей строке, если удается найти исходный текст программы с реализацией данной функции. В частности, внутрь функции printf не «зайти» не получится, и произойдет просто переход к следующей строке, так как нет соответствующего текста программы. Эта команда полезна для перехода внутрь функций, которые были разработаны, ведь текст программы тогда для них имеется.

Процесс отладки можно прервать досрочно, если нажать  (или выбрать Debug–>Stop debugger).

Если в режиме отладки консольное окно случайно было закрыто, то тогда надо нажать  (или выбрать Debug–>Stop debugger), чтобы можно было возобновить отладку.

С любого шага отладки можно запустить выполнение программы для оставшихся шагов без останова на каждом шаге. Для этого достаточно нажать  (или выбрать Debug–>Start, или нажать F8).

Также удобно использовать так называемые точки прерывания. Для установки точки прерывания достаточно нажать мышью на сером фоне справа от номера строки. Должен появиться красный круг, который и отмечает точку прерывания (рис. 55).

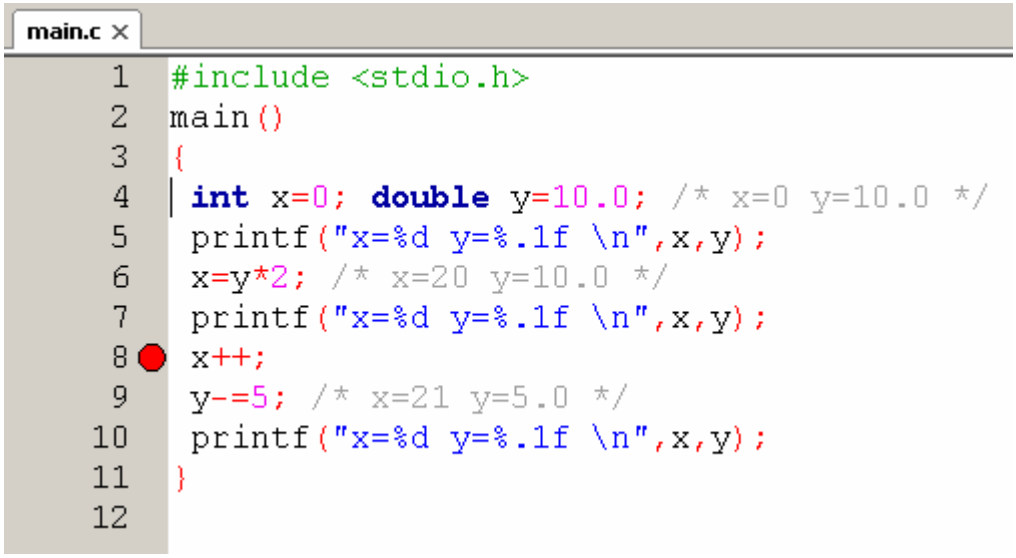

Рис. 55 – Установка точки прерывания для режима отладки

Теперь можно запустить процесс отладки, нажав  (или выбрав меню Debug–>Start, или нажав F8). Программа будет выполняться и остановится только на указанной точке прерывания, что отметит желтый треугольник (рис. 56).

main.c

```
1  #include <stdio.h>
2  main()
3  {
4   int x=0; double y=10.0; /* x=0 y=10.0 */
5   printf("x=%d y=%.1f \n",x,y);
6   x=y*2; /* x=20 y=10.0 */
7   printf("x=%d y=%.1f \n",x,y);
8   x++;
9   y-=5; /* x=21 y=5.0 */
10  printf("x=%d y=%.1f \n",x,y);
11 }
12
```

Рис. 56 – Приостановка программы на точке прерывания в режиме отладки

Теперь можно продолжать пошаговый процесс отладки или продолжить выполнение программы до конца.

Удобно в нужных (критических) местах программы поставить точки прерывания, чтобы не делать пошаговое выполнение, когда программа имеет достаточно большой размер. Тогда в точках прерывания можно будет оценить состояния переменных, чтобы убедиться в правильности выполнения программы.

Точку прерывания можно убрать, нажав на красный круг, соответствующий этой точке прерывания.

Также важной особенностью режима отладки является то, что значения переменных можно изменять *вручную*.

Например, пусть программа находится в следующем состоянии отладки (рис. 57).

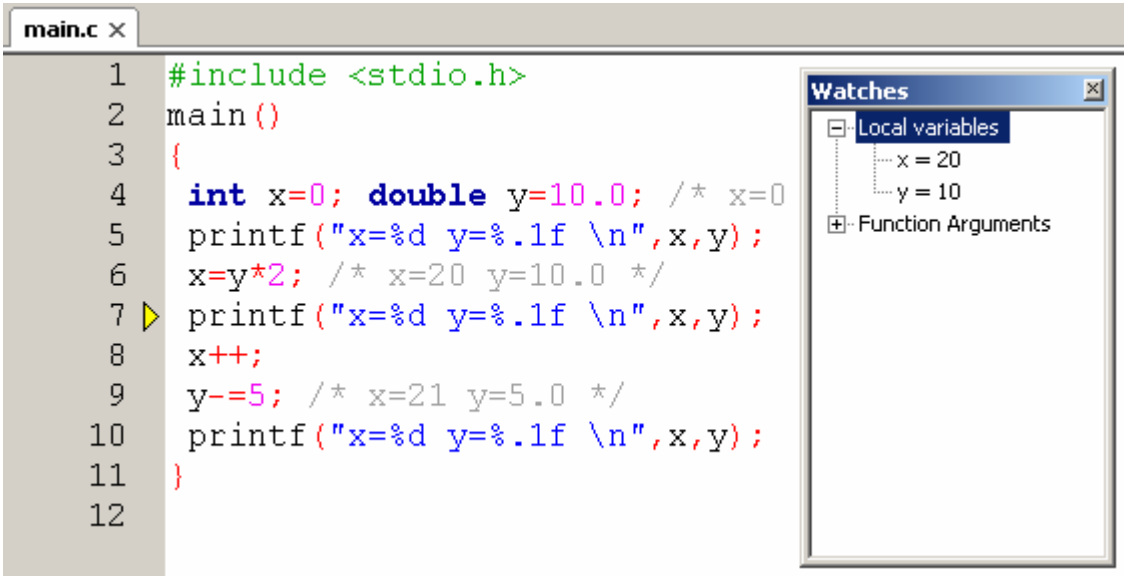

Рис. 57 – Индикация значений локальных переменных в ходе отладки

Если ничего не менять, то при переходе к следующему шагу в консольном окне будут выведены значения 20 и 10.0. Но значения переменных можно изменить с помощью окна Watches. Для этого надо нажать правой кнопкой мыши на переменной в окне Watches и из появившегося меню выбрать пункт Change value (рис. 58).

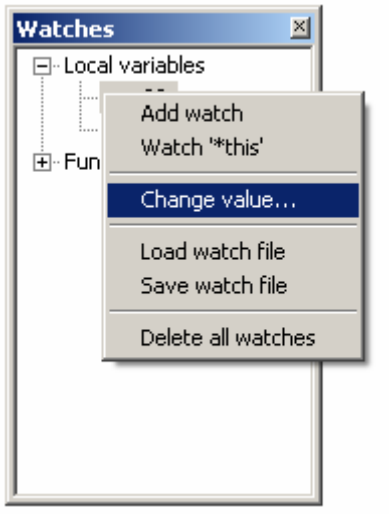

Рис. 58 – Запуск диалогового окна для изменения значения локальной переменной в ходе отладки

Появится окно, в котором можно ввести новое значение для переменной x. Надо будет нажать OK, тогда переменная примет новое значение. Именно это новое значение будет в дальнейшем выведено в консольное окно. Проверьте это.

Таким образом, отладка позволяет находить в программе ошибки и понять процесс выполнения программы. Используйте отладку, когда, например, программу выдает неправильный результат. Попробуйте пройти программу шаг за шагом (или останавливаясь в точках прерывания) чтобы проанализировать, как меняются переменные, как происходит переход от одной строки к другой строке программы. Это позволит найти ошибки и исправить программу.

2.1.4. Инструментальные программные средства поддержки проектных решений применительно к планированию задач реального времени

Ответом на сложности применения новых методов планирования в проектах систем управления (см. п. 1.4.7) стали следующие два направления практических внедрений моделей и методов планирования:

1) разработка исследовательских ОС реального времени и расширений реального времени для известных ОС (внедрение новых алгоритмов онлайнового планирования);

2) разработка инструментальных программных средств для поддержки инженерных решений в области планирования задач реального времени (внедрение новых моделей и методов планирования, характерных для стадии проектирования систем реального времени).

Первое направление стало ответом на медленные изменения в коммерческих ОС. Именно в исследовательских ОС внедряются и проверяются новые методы онлайнового планирования. В качестве примеров исследовательских ОС можно привести: Erika Enterprise [41], MaRTE OS[62],

S.Ha.R.K.[75]. В качестве примера использования исследовательской ОС реального времени см. работу [61], в которой описывается применение Erika Enterprise для системы управления беспилотным самолетом.

Другое направление для внедрения современных алгоритмов онлайнового планирования – это использование ядра Linux в качестве основы для создания системы реального времени. В качестве примера можно привести работу [42], где приводится описание дополнения к планировщику Linux, основанное на эффективном алгоритме планирования с динамическими приоритетами EDF. При этом обеспечивается планирование как периодических, так и апериодических задач. Также надо отметить развитие следующих проектов, связанных с расширением ядра Linux применительно к реальному времени: Xenomai [103], RTAI [73]. Важно, что эти проекты являются свободными, в частности бесплатными. Эти расширения используются в системах управления, и обеспечивают поддержку задач реального времени на системном уровне [40, 31]. В качестве примера можно привести программную архитектуру для приложений реального времени на базе Xenomai, используемую для управления служебным роботом, собирающим информацию из распределенной сенсорной сети и оперативно реагирующим на возникающие события [31]. Различные расширения Linux выглядят очень многообещающими с учетом того, что на рынке появляется большое количество дешевых, компактных и довольно мощных вычислительных платформ, ориентированных на Linux. В исследовании [67] обосновывается вывод о том, что применение Linux (вместе с расширениями реального времени) для промышленных устройств автоматизации имеет хорошую перспективу, даже с учетом широкого распространения коммерческих ОС реального времени.

Второе направление внедрения новых моделей и методов планирования связано с разработкой инструментальных программных средств для поддержки инженерных решений применительно к планированию задач

реального времени. Эти средства призваны помочь распространению новых методов планирования в инженерной практике. Они позволяют проводить анализ выполнимости, имитационное моделирование процессов выполнения задач и, в целом, решать различные проблемы планирования на этапах проектирования системы. В качестве примеров таких программных средств можно указать: TrueTime [96], Cheddar [89], SymTA/S [88], STORM [86], Romeo [72], chronVAL [33], chronSIM [32], RTSIM [74], RT-Druid [41].

В плане возможности проведения проектных и исследовательских работ по темам, находящимся на стыке планирования, автоматического управления и сетевых технологий, наиболее интересным инструментальным средством представляется TrueTime. Это средство позволяет одновременно моделировать: выполнение задач реального времени, поведение объектов при их автоматическом управлении, передачу данных в случае различных сетевых технологий. В качестве примера использования TrueTime можно привести работу [70]. Также на основе TrueTime можно проектировать и анализировать системы реального времени, использующие такие сложные методы планирования, как, например, планирование с обратными связями [49].

Еще можно особо выделить проект Cheddar, который, в частности, обеспечивает реализацию сложных моделей планирования, например, см. работу [80].

2.2. Использование системного программного обеспечения в составе управляющих систем реального времени

2.2.1. Типовые варианты использования системного программного обеспечения при разработке управляющих систем реального времени

Конечно, в большинстве случаев основным примером системного

ПО, используемым при разработке управляющих СРВ, является операционная система и различные утилиты, поставляемые вместе с операционной системой. Детали и особенности использования операционной системы излагаются в документации и различных справочных материалах, относящихся к данной операционной системе.

На базе данной операционной системы, как правило, имеется значительное количество утилит и вспомогательных программ, которые также относятся к системному ПО.

Помимо этого, важной особенностью управляющих СРВ является взаимодействие с внешними устройствами ввода-вывода (датчики, исполнительные устройства). Для связи с ними в операционной системе используются специальные программ, называемые драйверами.

При разработке ПО для микроконтроллеров часто возникает ситуация, что на микроконтроллере нет операционной системы. И из всего системного ПО на нем находится только загрузчик (bootloader). Пример – микроконтроллер семейства Arduino, обладающий подобным загрузчиком, который помогает устанавливать ПО на микроконтроллер. Поэтому при разработке ПО для микроконтроллеров может использоваться системное ПО в виде загрузчика ПО на этот микроконтроллер.

При *кросс-разработке* ПО для управляющих СРВ формируется двухмашинная конфигурация. В рамках этой конфигурации разработка ПО ведется на одном компьютере, а на второй (целевой) компьютер загружается уже откомпилированный код, и именно этот второй компьютер входит в состав управляющей СРВ. При таком подходе можно считать, что при разработке ПО управляющей СРВ используется системное ПО первого компьютера, на котором собственно и проходит разработка ПО. Его системное ПО обычно включает операционную систему, драйверы, вспомогательные программы, системы программирования.

2.2.2. Пример использования средств системного программного обеспечения для реализации работы с потоками

При разработке ПО для управляющей СРВ важно учитывать, что задачи РВ могут быть в виде потоков, реализуемых средствами этой операционной системы. Один из наиболее сложных аспектов использования системного ПО при разработке управляющих СРВ – это использование методов планирования и взаимодействия задач РВ, реализованных на базе потоков. Поэтому ниже рассмотрим пример использования потоков POSIX в рамках операционной системы на базе ядра Linux, а именно операционной системы Ubuntu 13.04 (в качестве простого примера).

2.2.2.1. Пример взаимодействия с устройствами вывода информации без использования потоков

Задачи РВ могут выполнять различные функции, например, чтение информации с датчиков и формирование управляющих команд, воздействующих на объект управления. Для того, чтобы нижеследующие примеры можно было выполнить без дополнительного оборудования (помимо обычного персонального компьютера), будут реализованы более простые варианты задач РВ. Однако, несмотря на свою простоту эти задачи позволят понять основные принципы работы с потоками

В качестве примера устройства вывода воспользуемся такими заметными объектами как светодиоды на клавиатуре. Важно, что эти объекты, как правило, уже имеются вместе с персональным компьютером. Так, на классической клавиатуре обычно имеется 3 светодиода, соответствующие Num Lock, Caps Lock, Scroll Lock. В случае ноутбука часть из этих светодиодов (или даже все) могут отсутствовать. В этом случае для выполнения этого примера надо подключить к ноутбуку клавиатуру.

Сначала попробуем управлять «вручную» этим светодиодами. Для этого нам понадобится утилита setleds. В дальнейшем предполагается ра-

бота в графической консоли. В командной строке, в графической консоли, последовательно выполняем (одновременно следя за светодиодами клавиатуры) следующие команды

```
sudo sh -c 'setleds -L +scroll < /dev/console'
sudo sh -c 'setleds -L -scroll < /dev/console'
```

После выполнения первой команды должен загореться светодиод, соответствующий Scroll Lock, после выполнения второй команды — он должен погаснуть.

Аналогичным образом, из комадной строки можно управлять другими светодиодами с помощью команд

```
sudo sh -c 'setleds -L +num < /dev/console'
sudo sh -c 'setleds -L -num < /dev/console'
sudo sh -c 'setleds -L +caps < /dev/console'
sudo sh -c 'setleds -L -caps < /dev/console'
```

Теперь напишем следующую простую программу на языке Си для проверки того, что управление светодиодами может выполняться.

```
#include <stdlib.h>
#include <unistd.h>
int main()
{
  while (1){
    system("setleds -L +num < /dev/console");
    system("setleds -L +caps < /dev/console");
    system("setleds -L +scroll < /dev/console");
    sleep(1);
    system("setleds -L +num < /dev/console");
    system("setleds -L +caps < /dev/console");
              system("setleds -L +scroll < /dev/console");
    sleep(1);
  }
  return 0;
}
```

Надо сделать сборку этой программы и запустить полученный файл с правами root. В дальнейшем предполагается, что все приведенные ниже программы, в которых используется setleds, запускаются с правами root.

В этой программе с помощью функции system вызываются ранее рассмотренные команды на основе setleds, которые управляют светодиодами. После блока команд на включение светодиодов и блока команд на их выключение вызывается функция sleep с аргументом 1, что обеспечивает задержку в 1 секунду. Все это располагается в бесконечном цикле while. В итоге, при запуске программы происходит мигание всех трех светодиодов с периодом в 2 секунды. Так как используется бесконечный цикл, поэтому выход из программы осуществляется принудительным прерыванием ее выполнения, например нажатием Ctrl-C.

Чтобы реализовать переключение светодиодов с меньшим периодом нужны дробные значения секунды. Поэтому лучше в программе вызовы функции sleep заменить на вызовы функции usleep, аргумент которой задает время в микросекундах. Например, вместо sleep(1) надо поместить usleep(1000000) для того, чтобы период не изменился. Поэкспериментируйте со значением периода. Например, можно определить, при какой частоте мигания светодиодов уже нельзя будет различить отдельные включения и выключения.

Теперь дополним немного программу, чтобы сделать удобнее ее в смысле выполнения и управления. Нам понадобится библиотека ncurses, и для ее установки необходимо иметь установленным пакет с названием вида lib64ncurses5-dev (естественно, что для других дистрибутивов и других версий название пакета может немного меняться). В дальнейшем предполагается, что при указании необходимости того или иного пакета, надо его установить, если он еще не был установлен.

Также надо добавить ключ -lncurses для компоновщика, чтобы обеспечить правильную сборку.

После этого можно запустить следующую программу

```
#include <stdlib.h>
#include <unistd.h>
#include <ncurses.h>
int t=100000;
void run_leds()
{
  system("setleds -L +num < /dev/console");
  usleep(t);
  system("setleds -L +caps < /dev/console");
  usleep(t);
  system("setleds -L +scroll < /dev/console");
  usleep(t);
  system("setleds -L -num < /dev/console");
  usleep(t);
  system("setleds -L -caps < /dev/console");
  usleep(t);
  system("setleds -L -scroll < /dev/console");
  usleep(t);
}
int main()
{
  int key;
  initscr();
  raw();
  noecho();
  keypad(stdscr, TRUE);
  while(key!=27){
    move(0,0);
    printw("%d\n",t);
    run_leds();
    timeout(0);
    key=getch();
    flushinp();
```

```
    if((key==KEY_UP)&&(t>=10000)) t-=10000;
    if((key==KEY_DOWN)&&(t<1000000)) t+=10000;
  }
  endwin();
  return 0;
}
```

Эта программа реализует «бегущие огни» на основе трех клавиатурных светодиодов. При этом с помощью клавиш «вверх» и «вниз» можно изменять скорость этих «бегущих огней», и в окне отображается значение переменной t, которая определяет эту скорость. Выход из программы выполняется нажатием клавиши Esc.

В этой программе реализована функция run_leds(), которая выполняет один цикл «бегущих огней», учитывая значение глобальной переменной t.

Попробовав поработать с этой программой, нетрудно заметить ее недостаток, состоящий в том, что клавиши «вверх», «вниз» и Esc отрабатываются не сразу же, а после того, как выполнится полный цикл «бегущих огней» (в момент, когда они все гаснут). Это особенно заметно при маленькой скорости «бегущих огней». Вполне понятно, почему это происходит, учитывая, что считывание нажатой клавиши (timeout(0) и key=getch()) выполняется после вызова run_leds(), то есть после выполнения одного цикла «бегущих огней».

Можно конечно, поставить строки для проверки нажатости клавиш после каждой команды управления светодиодов. Реакция программы станет быстрее, но все равно в худшем случае она может достигать текущего значения t.

Есть еще способы улучшить реакцию программы на нажатие клавиш. Они предполагают замену usleep(t) на цикл, в котором одновременно выполняется задержка и проверяется нажатость клавиши. В данном простом примере это сделать возможно, так как команда xset, выполняемая с помо-

щью функции system, здесь выполняется достаточно быстро. Но представьте, если бы эта команда выполнялась бы достаточно долго. Тогда бы все равно пришлось ожидать ее окончания, не реагируя на нажатие клавиши.

Какой же выход? Как можно улучшить реакцию программы на нажатие клавиш, не изменяя функциональности программы, то есть обеспечивая тот же алгоритм работы «бегущих огней»?

Выход есть, и он связан с использованием потоков.

2.2.2.2. Пример взаимодействия с устройствами вывода информации на основе использования одного потока

В предыдущей программе функция run_leds() вызывалась из основного цикла. Надо попробовать вынести реализацию «бегущих огней» из основного цикла программы в отдельный поток. Тогда в основном цикле останется только проверка нажатия клавиши и отработка этих нажатий.

Для этого после функции run_leds() перед main() добавим следующую функцию

```
void *thread_leds(void* arg)
{
  while(1){
    run_leds();
  }
  return NULL;
}
```

Эта функция как раз и определяет поток, в котором выполняется бесконечный цикл «бегущих огней». Теперь еще надо изменить функцию main() следующим образом. Во первых, надо убрать строки

```
    run_leds();
    timeout(0);
```

Они теперь больше не нужны, так как «бегущие огни» будут реализованы в потоке.

Во-вторых, в самом начале функции main() надо добавить строки

```
pthread_t thr;
pthread_create(&thr, NULL, thread_leds, NULL);
```

Эти строки создают поток thread_leds (обратите внимание, что это название функции, которую мы добавили выше).

Осталось в первую строку файла добавить

```
#include <pthread.h>
```

Также надо добавить ключ -lpthread для компоновщика, чтобы обеспечить правильную сборку.

Запустите программу и заметьте, как сейчас быстро (практически без видимой задержки) отрабатывается нажатие клавиш «вверх», «вниз», и при этом соответствующим образом меняется число, определяющее задержку «бегущих огней».

Это происходит из-за того, что поток с «бегущими огнями» выполняется параллельно с основным циклом. Таким образом, программа не дожидается окончания очередного цикла «бегущих огней» (ведь они выполняются теперь параллельно), а практически сразу считывает нажатие клавиши.

Но нетрудно заметить недостаток в работе этой программы, состоящий в том, что после выхода из программы часть светодиодов (а иногда и все светодиоды) остаются включенными. Но ведь функция run_leds() завершается тем, что все светодиоды отключаются. Почему же тогда это происходит?

Все дело в том, что при нажатии клавиши Esc происходит выход из программы, и при завершении основной части программы (функции main()) текущие потоки закрываются, прерывая свое выполнение «на полуслове». Как сделать так, чтобы можно было дождаться окончания выполнения потока?

Ну, для начала надо организовать поток так, чтобы он завершался. В

текущей реализации используется бесконечный цикл while(1). Надо сделать так, чтобы поток «знал», что пора завершаться. Для перед функцией потока запишем строку с объявлением глобальной переменной

```
int stop_leds=0;
```

Эта переменная изначально делается равной нулю. Кроме этого внутри функции потока строка

```
while(1){
```

меняется на строку

```
  while(!stop_leds){
```

Теперь чтобы прекратить выполнение потока, достаточно установить переменную stop_leds в ненулевое значение. После строки с endwin() добавим строку

```
  stop_leds=1;
```

Теперь после выхода из основного цикла по нажатию Esc потоку будет посылаться команда о завершении (с помощью переменной stop_leds).

Но если проверить работу программы, то окажется, что ничего не поменялось (светодиоды могут оставаться включенными).

Это происходит из-за того, что после строки, содержащей stop_leds=1, основная программа заканчивается раньше, чем заканчивается поток. Надо сделать так, чтобы основная программа дожидалась окончания основного потока.

Это реализуется с помощью функции pthread_join (см. п. ???), а именно после строки, содержащей stop_leds=1, надо добавить строку

```
  pthread_join(thr,NULL);
```

Теперь программа должна работать правильно, то есть после выхода из программы все светодиоды должны быть выключены.

При этом заметьте, что если делать «бегущие огни» медленными, увеличив задержку с помощью клавишу «вниз», то выход из программы может удлиниться. Это как раз происходит из-за того, что основная программа ожидает окончания основного потока. И процесс окончания потока

заметен по тому, что «бегущие огни» заканчивают свой цикл, и как только они заканчивают этот цикл, программа завершается.

2.2.2.3. Пример взаимодействия с устройствами вывода информации на основе использования нескольких потоков

Организуем три потока, по одному потоку на каждый светодиод, и сделаем независимое управление частотой мигания каждого светодиода.

Для этого создадим соответствующую программу. В начальной части выполним подключение заголовочных файлов, а также объявление глобальных переменных.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <ncurses.h>
int t_num=100000;
int t_caps=90000;
int t_scroll=80000;
int stop_leds=0;
```

Здесь имеется три переменные определяющие задержки (полупериод), используемые для мигания светодиодами. Они сделаны разными (почему будет понятно позднее). Также объявлена переменная stop_leds, необходимость в которой была рассмотрена ранее.

Далее определим три функции для трех потоков, соответствующих трем светодиодам.

Функция для потока, реализующего мигание светодиодом, который соответствует Num Lock.

```
void *thread_led_num(void* arg)
{
  while(!stop_leds){
    system("setleds -L +num < /dev/console");
    usleep(t_num);
```

```
    system("setleds -L -num < /dev/console");
    usleep(t_num);
  }
  return NULL;
}
```

Функция для потока, реализующего мигание светодиодом, который соответствует Caps Lock.

```
void *thread_led_caps(void* arg)
{
  while(!stop_leds){
    system("setleds -L +caps < /dev/console");
    usleep(t_caps);
    system("setleds -L -caps < /dev/console");
    usleep(t_caps);
  }
  return NULL;
}
```

Функция для потока, реализующего мигание светодиодом, который соответствует Scroll Lock.

```
void *thread_led_scroll(void* arg)
{
  while(!stop_leds){
    system("setleds -L +scroll < /dev/console");
    usleep(t_scroll);
    system("setleds -L -scroll < /dev/console");
    usleep(t_scroll);
  }
  return NULL;
}
```

Здесь важно отметить, что в каждой функции используется соответствующая переменная задержки (полупериода).

В начале основной функции программы (main) создается три потока с использованием только что определенных функций.

```
int main()
{
  pthread_t th_num, th_caps, th_scroll;
  pthread_create(&th_num, NULL, thread_led_num, NULL);
  pthread_create(&th_caps, NULL, thread_led_caps, NULL);
  pthread_create(&th_scroll,   NULL,   thread_led_scroll,
NULL);
```

Затем, выполняется стандартный цикл опроса клавиатуры.

```
  int key;
  initscr();
  raw();
  noecho();
  keypad(stdscr, TRUE);
  while(key!=27){
    move(0,0);
    printw("%d\t%d\t%d\n",t_num, t_caps, t_scroll);
    key=getch();
    flushinp();
    if((key==KEY_IC)&&(t_num>=10000)) t_num-=10000;
    if((key==KEY_DC)&&(t_num<1000000)) t_num+=10000;
    if((key==KEY_HOME)&&(t_caps>=10000)) t_caps-=10000;
    if((key==KEY_END)&&(t_caps<1000000)) t_caps+=10000;
    if((key==KEY_PPAGE)&&(t_scroll>=10000))
      t_scroll-=10000;
    if((key==KEY_NPAGE)&&(t_scroll<1000000))
      t_scroll+=10000;
  }
```

При этом для управления скоростью мигания каждым светодиодом используется своя пара клавиш. Для светодиода, соответствующего Num Lock, используются клавиши Insert и Delete (с кодами KEY_IC, KEY_DC). Для светодиода, соответствующего Caps Lock, используются клавиши Home и End (с кодами KEY_HOME, KEY_END). Для светодиода, соответствующего Scroll Lock, используются клавиши Page Up и Page Down (с ко-

дами KEY_PPAGE, KEY_NPAGE).

В заключительной части программы, которая выполняется при выходе из основного цикла, происходит выход формирование команды остановки потоков на основе задания значения 1 для переменой stop_leds, а также выполняется ожидание окончания всех трех потоков с помощью функции pthread_join.

```
    endwin();
    stop_leds=1;
    pthread_join(th_num,NULL);
    pthread_join(th_caps,NULL);
    pthread_join(th_scroll,NULL);
    return 0;
}
```

Программу, как и прежде, надо запускать с правами root. При выполнении программы можно управлять скоростью мигания каждого светодиода с помощью соответствующей пары клавиш. Нетрудно убедиться, что светодиоды управляются независимо друг от друга. При этом на экран выводятся три числа, каждое из которых определяет полупериод мигания соответствующего светодиода.

Можно заметить, что функции thread_led_num, thread_led_caps, thread_led_scroll, используемые потоками, очень похожи. Можно ли реализовать три потока на основе определения одной функции? Да, это можно сделать, используя различные значения аргумента этой функции.

Чтобы проверить это, сначала заменим определение переменых t_num, t_caps, t_scroll на определение массива

```
int t[NUM_LEDS]={100000,90000,80000};
```

В этом определении используется константа NUM_LEDS, поэтому выше она должна быть задана строкой

```
#define NUM_LEDS 3
```

Также добавим массив строчных кодов светодиодов

```
char *led_name[NUM_LEDS]={"num","caps","scroll"};
```

Теперь заменим в тексте программы определения функций hread_led_num, thread_led_caps, thread_led_scroll на определение следующей функции.

```
void *thread_led(void* arg)
{
  int led_id=*(int*)arg;
  if ((led_id <0)||(led_id>NUM_LEDS-1))
    return NULL;
  char command_led_on[35];
  char command_led_off[35];
  sprintf(command_led_on,
    "setleds -L +%s < /dev/console", led_name[led_id]);
  sprintf(command_led_off,
    "setleds -L -%s < /dev/console", led_name[led_id]);
  while(!stop_leds){
    system(command_led_on);
    usleep(t[led_id]);
    system(command_led_off);
    usleep(t[led_id]);
  }
  return NULL;
}
```

В этой функции переменная led_id получаем номер (от 0 до 2), который передается через указатель arg. С учетом ранее сформированных массивов значение led_id соответствует одному из трех светодиодов. Далее в функции проверяется, что значение led_id находится в разрешенном диапазоне. Затем формируются строки команд для функции system на основе значения элемента массива led_name. После этого выполняется цикл, в котором задержка определяется нужным элементом массива t.

В основной функции main надо вместо трех строк создания потоков поместить строки

```
int id[]={0,1,2};
pthread_create(&th_num, NULL, thread_led,
 (void *)&id[0]);
pthread_create(&th_caps, NULL, thread_led,
  (void *)&id[1]);
pthread_create(&th_scroll, NULL, thread_led,
  (void *)&id[2]);
```

Если раньше четвертым аргументом pthread_create было значение NULL, то теперь это указатель на один из элементов массива id, в котором находятся «номера» потоков. Массив сделан для общности, так как используя данный принцип, например, можно сделать цикл, в котором заполнять его элементы, и потом передавать функции pthread_create указатель на нужный элемент.

Также ниже все упоминания переменных t_num, t_caps, t_scroll надо заменить на t[0], t[1], t[2].

В итоге программа должна работать так же. Но преимущество ее состоит в том, что если мы захотим изменить работу со светодиодом в потоке, то это надо будет делать не в трех разных функциях, а в одной.

Можно пойти дальшей и вместо трех переменных th_num, th_caps, th_scroll сделать массив и организовать цикл, в котором выполняется создание нужных потоков. Это оставляется читателю в качестве упражнения. Реализация подобных циклов по созданию потоков и использование одной функции с разными аргументами для потока позволяет формировать большое множество потоков без увеличения числа определяемых функций.

2.2.2.4. Пример работы с мьютексами при использовании нескольких потоков

Попробуйте в рассматриваемой программе заменить строку

```
int t[NUM_LEDS]={100000,90000,80000};
```

на строку

```
int t[NUM_LEDS]={100000,100000,100000};
```

то есть сделать так, что при старте все светодиоды имели одинаковый период.

Запустив программу, можно заметить, что светодиоды начинают мигать с переменными интервалами, и вести себя как-то странно. Хотя они должны мигать синхронно.

Если начальные значения элементов массива t сделать в 10 раз больше (увеличить начальный период), то можно заметить, что иногда тот или иной светодиод не включается вместе со всеми, или не выключается вовремя.

Если же при этом значения периодов изменить и сделать их неравными, но почти одинаковыми, то подобных проблем не почти возникает.

Почему все это происходит?

Все дело в том, что при изначально равных периодах все три потока практическ в один и тот же момент вызывают пытаются выполнить переключение светодиода из одного состояние в другое. Для этого происходит вызов функции system с нужной командной строкой. И хотя функция system выполняется быстро, все равно часто получается так, что, упрощенно говоря, два похожих обращения к светодиодам «перекрываются», приводя к ошибками при переключении. В итоге, оказывается, что какой-то светодиод не переключается в нужное состояние. Когда же периоды немного различаются, то такие ситуации почти не возникают, и подобных нарушений не возникает.

Как же избавиться от подобных ситуаций при равных периодах?

Здесь поможет механизм мьютекса (mutex). Этот механизм позволяет исключить одновременное выполнение тех или иных частей программного кода. В данном случае нам надо исключить одновременное выполнение вызовов функции system внутри потоков. Для этого сначала создадим переменную для мьютекса, поместив перед функцией потока следующую

строку

```
pthread_mutex_t lock;
```

Теперь имеем переменную-мьютекс lock. И остается заменить строку

```
system(command_led_on);
```

на три строки

```
pthread_mutex_lock(&lock);
system(command_led_on);
pthread_mutex_unlock(&lock);
```

Аналогично строка

```
system(command_led_off);
```

заменяется на три строки

```
pthread_mutex_lock(&lock);
system(command_led_off);
pthread_mutex_unlock(&lock);
```

Теперь попробуйте выполнить программу и проверьте, что теперь светодиоды начинаются мигать синхронно даже при одинаковых начальных периодах.

Это происходит из-за того, что если какой-то процесс первым подходит к команде включения или выключения своего светодиода, то перед этим он «закрывает» мьютекс lock. И это не позволяет другим потокам выполнить команду через функцию system, так как в их функции тоже имеются вызовы pthread_mutex_lock и в них используется тот же мьютекс lock. И эти потоки доходят до вызова pthread_mutex_lock(&lock) и начинают ожидать «открытия» мьютекса lock. После того как первый из упомянутых потоков выполнил свой вызов функции system, он «открывает» мьютекс lock, что позволяет одному из оставшихся потоков начать выполнение вызова функции system. Но этот поток аналогичным образом «закроет» lock, не позволяя третьему потоку начать вызов system. В итоге, в случае, когда начальные периоды равны, то вместо одновременного вызова функции system происходит из последовательный вызов. Но поскольку вызов функции

system выполняется очень быстро, то кажется, как будто все светодиоды меняют свое состояние одновременно.

Этот простейший пример показывает, как мьютексы позволяют реализовать взаимное исключение. В данном примере получается, что один поток исключает одновременное с ним выполнение вызова функции system другими аналогичными потоками. В данном примере именно вызовы функции system рассматриваются в качестве критических секций кода, которые защищаются мьютексом lock.

2.3. Разработка системного программного обеспечения в процессе разработки управляющих систем реального времени

2.3.1. Необходимость разработки системного программного обеспечения в процессе разработки управляющих систем реального времени

Обычно процесс разработки системного ПО отделен от разработки управляющей СРВ. Это касается в первую очередь разработки операционных систем, систем программирования, вспомогательных программ (утилит) в составе операционной системе.

Однако при разработке управляющих СРВ может потребоваться разработка драйвера для устройства, которое не поддерживается используемой операционной системой (для которого не существует драйвера).

Также в ряде случаев может понадобится разработать собственный загрузчик для вычислительного устройства, которое работает без использования операционной системы.

Для прояснения некоторых особенностей разработки драйверов, рассмотрим следующий пример, относящийся к области разработки драйверов (модулей) для операционной системы на базе ядра Linux. В качестве при-

мера операционной системы используется Ubuntu 13.04.

2.3.2. Пример разработки драйвера

Сначала устанавливаем заголовочные файлы для ядра, для этого устанавливаем пакет командой

```
sudo apt-get install linux-headers-$(uname -r)
```

Теперь в создаем файл с именем my_driver.c. Сначала этот файл будет очень простым, лишь для проверки того, что загружается и выводить сообщения, а именно он должен содержать

```
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");
int init_module()
{
  printk(KERN_INFO "init my_driver\n");
  return 0;
}
void cleanup_module()
{
  printk(KERN_INFO "cleanup my_driver\n");
}
```

Этот модуль ничего не делает. Он лишь служит для проверки механизма установки и выгрузки модуля.

Теперь в том же каталоге, где расположен файл my_driver.c создаем файл с именем

```
Makefile
```

и следующим содержимым

```
obj-m += my_driver.o
KDIR ?= /lib/modules/$(shell uname -r)/build
all:
make -C $(KDIR) M=$(PWD) modules
clean:
```

```
make -C $(KDIR) M=$(PWD) clean
```

Важно, чтобы в этом файле строка, следующая после all, и строка, следующая после clean, начинались с табуляции.

Теперь, находясь в этом же каталоге, выполняем команду

```
make
```

В итоге, в данном каталоге должно сформироваться много разных файлов. Нас в первую очередь интересует файл модуля my_driver.ko. Устанавливаем этот модуль командой

```
sudo insmod my_driver.ko
```

Теперь он установлен, для этого достаточно выполнить команду lsmod, будет выведен список установленных модулей, среди которых должен быть my_driver. Также можно посмотреть список сообщения ядра командой

```
dmesg
```

В выведенном списке в конце можно увидеть фразу «init my_driver». Эта фраза выводится модулем с помощью функции printk в ходе установки (см. программу данного модуля).

Как и было сказано, это пока что простейший вариант модуля, он ничего не делает. Но его можно выгрузить командой

```
sudo rmmod my_driver
```

Заметьте, что теперь этого модуля уже нет в списке всех модулей, а также в списке сообщения ядра появляется фраза «cleanup my_driver».

Теперь дополним этот драйвер так, чтобы он поддерживал работу символьного устройства (Character Device) и можно было проследить чтение и запись в это устройство.

В начале файла добавляем строку

```
#define DEVICE_NAME "my_driver"
```

Эта строка будет определять имя драйвера (модуля) и мы ее будем использоваться неоднократно, поэтому лучше добавить это макроопределение, чтобы потом при необходимости можно было легко поменять имя

драйвера.

Добавляем дополнительный заголовочный файл

```
#include <linux/fs.h>
```

В текст нашего драйвера (модуля) перед функцией init_module добавляем описание структуры соотвествия файловых операций

```
struct file_operations fops = {
  .read=device_read,
  .write=device_write,
  .open=device_open,
  .release=device_release
};
```

В этой структуре указываются имена функций, которые будут вызываться в случае указанных четырех файловых операций. И эти функции мы должны определить в модуле.

Перед функцией init_module также добавляем строки

```
static unsigned int major;
static int device_opened=0;
```

Здесь переменная major служит для хранения так называемого старшего номера устройства. Этот номер нужен для того, что определять соответствие между драйвером и устройством. Переменная device_opened служит в качестве счетчика того, сколько раз открыто устройство.

В связи с добавлением этих переменных изменяем функцию init_module следующим образом

```
int init_module()
{
  major = register_chrdev(0, DEVICE_NAME, &fops);
  if(major < 0) {
    printk(KERN_INFO "%s: failed with: %d\n",
      DEVICE_NAME, major);
    return major;
  }
  printk(KERN_INFO "%s: added with: %d\n",
```

```
    DEVICE_NAME, major);
  return 0;
}
```

Здесь добавилась регистрация символьного устройства с именем DEVICE_NAME и получением номера в переменную major. Также проверяется успешность этой регистрации с выводом соответствующего сообщения.

Также изменяем функцию cleanup_module следующим образом

```
void cleanup_module()
{
  unregister_chrdev(major, DEVICE_NAME);
  printk(KERN_INFO "%s: removed\n", DEVICE_NAME);
}
```

Здесь выполняется удаление устройства из регистра с выводом сообщения об этом.

Перейдем теперь к функции открытия устройства. Для этого добавляем в текст драйвера определение этой функции.

```
static int device_open(struct inode *inode,
                       struct file *file)
{
  if (device_opened)
    return -EBUSY;
  device_opened++;
  try_module_get(THIS_MODULE);
  return 0;
}
```

В этой функции проверяется, было ли уже открыто это устройство. Если оно уже было открыто, то функция возвращает константу ошибки. Иначе (если еще не было открыто) устанавливается признак в переменной об открытии, а также с помощью функции try_module_get увеличивает на единицу счетчик открытия. Этот счетчик проверяется при попытке удале-

ния модуля. И если устройство было открыто, то модуль не будет удален.

Аналогично добавляю функцию закрытия устройства.

```
static int device_release(struct inode *inode,
                          struct file *file)
{
  device_opened--;
  module_put(THIS_MODULE);
  return 0;
}
```

Здесь выполняется уменьшения счетчика device_opened, также с помощью функции try_module_get уменьшается на единицу счетчик открытия.

Теперь добавляем функцию чтения, она для начала будет пустой, и будет выполнять роль заглушки (для проверки).

```
static ssize_t device_read(struct file *filp,
                           char *buffer,
                           size_t length,
                           loff_t * offset)
{
  printk(KERN_INFO "%s: read\n", DEVICE_NAME);
  return 0;
}
```

Сейчас эта функция (заглушка) просто сообщает о попытке чтения формированием сообщения с помощью функции printk.

Аналогично, добавляем функцию записи (в виде заглушки).

```
static ssize_t device_write(struct file *filp,
                            const char *buff,
                            size_t len,
                            loff_t * off)
{
  printk(KERN_INFO "%s: write\n", DEVICE_NAME);
  return len;
```

```
}
```

Сейчас эта функция (заглушка) просто сообщает о попытке записи формированием сообщения с помощью функции printk.

Еще остается в верхнюю часть файла, перед структурой fops, разместить объявления функций, то есть надо добавить туда следующие строки

```
static int device_open(struct inode *inode,
                       struct file *file);
static int device_release(struct inode *inode,
                          struct file *file);
static ssize_t device_read(struct file *filp,
                           char *buffer,
                           size_t length,
                           loff_t * offset);
static ssize_t device_write(struct file *filp,
                            const char *buff,
                            size_t len,
                            loff_t * off);
```

Теперь собираем модуль, выполняя команду make, и проверяем его установку и выгрузку с помощью команд sudo insmod и sudo rmmod, как это было сделано ранее. Также важно увидеть в списке сообщений ядра, что выводится сообщение об добавлении и удалении модуля с указанием старшего номера устройства.

Установите данный модуль с помощью команды sudo insmod.

Создадим файл устройства для данного модуля. Для этого выполняем команду вида

```
sudo mknod /dev/my_driver c N 0
```

где вместо N надо указать старший номер устройства (он выводится в списке сообщений ядра с помощью соответствующего сообщения). После успешного выполнения данной команды появляется файл /dev/my_driver в который можно выполнять запись, и из которого можно читать. Для удобства дадим всем права на чтение и запись для этого файла командой

```
sudo chmod a+rw /dev/my_driver
```

Выполните чтение из созданного устройства командой

```
cat /dev/my_driver
```

и проверьте список сообщения ядра. Там должна быть выведена строка

```
my_driver: read
```

После каждой операции чтения из /dev/my_driver должно выводиться это сообщение. Данное сообщение выводится с помощью вызова функции printk из функции device_read нашего модуля.

Теперь попробуем запись в это устройство с помощью команды

```
echo 111 > /dev/my_driver
```

Здесь выполняется запись строки «111», хотя вообще-то не важно, что пытаться записывать, важен сам факт записи.

Проверяем (через dmesg) список сообщений ядра, в конце списка должна быть выведена строка

```
my_driver: write
```

После каждой операции записи в /dev/my_driver должно выводиться это сообщение. Данное сообщение выводится с помощью вызова функции printk из функции device_write нашего модуля.

Таким образом, реализован прототип драйвера для символьного устройства. В данном простом примере символьное устройство только сообщает о попытке прочитать его и о попытке записать в него. В дальнейшем эта заготовка драйвера может использоваться для реализации более сложных драйверов, взаимодействующих с реальными устройствами ввода-вывода.

2.3.3. Разработка драйвера для устройства ввода-вывода

В качестве примера реального устройства ввода-вывода рассмотрим плату ввода-вывода 5700 Octagon Micro PC. Данная плата ввода-вывода присоединяется к управляющему компьютеру на основе шины ISA. Струк-

турная схема этого устройства представлена на рис. ???.

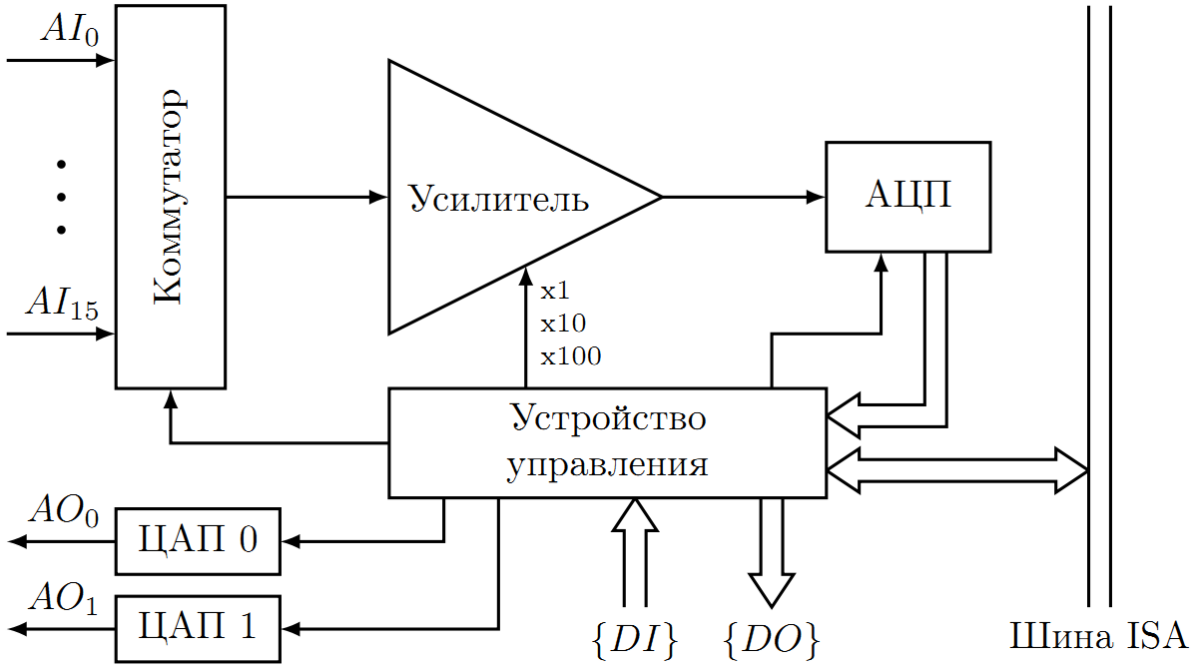

Рис. ???. Структурная схема платы ввода-вывода 5700 Octagon Micro PC

У нее имеется 16 коммутируемых каналов ввода аналоговых сигналов (AI_0 …, AI_{15}). Устройство управления, находящееся в составе платы, выбирает канал с помощью коммутатора. При коммутации каждого из этих каналов сигнал может быть усилен. Один из трех коэффициентов усиления (x1, x10 ,x100) устанавливается с помощью устройства управления.

Также устройство управления может выполнять настройку аналого-цифрового преобразователя (АЦП). В частности, может быть задана автокомпенсация нуля, а также автокалибровка. Кроме того, устройство управления выдает команду начала аналого-цифрового преобразования. АЦП является 12-ти разрядным (плюс еще один знаковый разряд). Диапазон входных значений АЦП: [-5,+5]В. Время одного преобразования составляет примерно 15 мкс. При включении автокомпенсации нуля это время увеличивается примерно в 2 раза.

Цифровой код, полученный после аналого-цифрового преобразования, поступает в соответствующий регистр устройства управления. Этот

код посредством шины ISA может быть прочитан для дальнейшей его обработки управляющим компьютером.

Также плата имеет два аналоговых канала вывода, реализованных на основе двух независимых цифро-аналоговых преобразователей (ЦАП). Диапазоны выходных сигналов (AO_0, AO_1) задаются специальными перемычками на плате. Можно устанавливать следующие диапазоны: [0,+5]В, [0,+10]В, [-5,+5]В.

Кроме того, предоставляется возможность для работы с дискретными каналами ввода-вывода $\{DI\}$, $\{DO\}$.

Плата ввода-вывода обладает хорошими эксплуатационными характеристиками. В частности, она имеет: рабочий температурный диапазон от -40°С до +85°С; устойчивость к вибрациям до 5g и к ударам до 20g.

Однако, при использовании операционной системы семейства Linux для реализации обмена данными с этим устройством необходимо разработать соответствующий драйвер.

В качестве примера напишем драйвер для символьного устройства, обеспечивающий работу с данной платой ввода-вывода. В ходе реализации этого драйвера будет обеспечена возможность имитации работы платы ввода-вывода, что позволит проверять работу драйвера даже при отсутствии устройства ввода-вывода.

Для удобства работы с новым драйвером создаем новый каталог, в который копируем ранее созданный файл my_driver.c. Переименуем этот файл так, чтобы теперь он назывался micropc5700.c (имя файла соответствует названию устройства, для которого разрабатывается драйвер).

Теперь в этом же каталоге создаем файл с именем

`Makefile`

и следующим содержимым

```
obj-m += micropc5700.o
KDIR ?= /lib/modules/$(shell uname -r)/build
```

```
all:
make -C $(KDIR) M=$(PWD) modules
clean:
make -C $(KDIR) M=$(PWD) clean
```

Важно, чтобы в этом файле строка, следующая после all, и строка, следующая после clean, начинались с табуляции.

Теперь, находясь в этом же каталоге, выполняем команду

```
make
```

Тем самым, проверяем, что сборка драйвера выполняется правильно. При этом в каталоге должно сформироваться множество файлов, в том числе и файл модуля microрс5700.ko. Можно проверить работу этого модуля, он должен выполнять функции ранее разработанного примера драйвера, так как в качестве исходного текста сейчас используется текст из файла my_driver.c.

Начнем менять и дополнять содержимое файла microрс5700.c так, чтобы в итоге получить драйвер для заданного устройства ввода-вывода.

Для начала поменяем первую строку на строку

```
#define DEVICE_NAME "microрс5700"
```

так как имя драйвера и имя устройства поменялись.

Особенностью данного устройства ввода-вывода является то, что для работы с ним необходимо иметь доступ к портам ввода-вывода. Адреса этих портов формируют диапазон из 12-ти портов, начиная с так называемого базового адреса. Значение базового адреса задается перемычкой на плате ввода-вывода. По умолчанию, это значение равно 0x100. В случае возможности его изменения надо будет обеспечивать механизм, с помощью которого можно было бы задавать эти изменения. Сейчас для простоты будем считать, что значение базового адреса является постоянным и равно 0x100. Поэтому в верхнюю часть файла исходного текста драйвера добавим следующие строки

```
#define BASE_ADDRESS 0x100
```

```
#define PORT_START BASE_ADDRESS
#define PORT_COUNT 12
```

В дальнейшем, значение базового адреса будет обозначаться BASE_ADDRESS.

Также перед определением функции init_module размещаем следующую строку

```
static int emu_mode = 1;
```

Эта переменная понадобится в дальнейшем для реализации режима имитации платы ввода-вывода. В случае работы с реальным устройством ввода-вывода надо поменять начальное значение переменной emu_mode на значение 0, чтобы отключить режим имитации платы ввода-вывода.

Драйвер будет работать с портами ввода-вывода, поэтому в функции init_module надо реализовать выделение используемого диапазона портов ввода-вывода. Этот диапазон начинается с порта с адресом PORT_START и занимает адресное пространство, соответствующее заданному количеству портов, которое равно PORT_COUNT. Учитывая это, в функцию init_module перед строкой, содержащей «return 0;», добавляем фрагмент

```
if (check_region(PORT_START, PORT_COUNT))
{
  emu_mode = 1;
  printk("%s: ports [%#x, %#x] allocation failed\n",
         DEVICE_NAME, PORT_START,
         PORT_START+PORT_COUNT-1);
} else {
  request_region(PORT_START, PORT_COUNT, DEVICE_NAME);
  printk ("%s: allocated i/o ports [%#x, %#x]\n",
          DEVICE_NAME, PORT_START,
          PORT_START+PORT_COUNT-1);
}
```

В этом фрагменте сначала с помощью функции check_region проверяется возможность выделить требуемый диапазон портов. Если оказыва-

ется, что это сделать нельзя, то тогда переменной emu_mode принудительно устанавливается значение 1, что соответствует включению режима имитации платы ввода-вывода, а также формируется сообщение с помощью функции printk. Иначе выделяется требуемый диапазон портов с помощью функции request_region, и выводится сообщении о выделении этого диапазона портов.

Еще надо сделать так, чтобы при выгрузке модуля осуществлялось освобождение ранее выделенного диапазона портов. Для этого в функцию cleanup_module в качестве первой строки добавим строку

```
release_region(PORT_START, PORT_COUNT);
```

В этой строке функция release_region обеспечивает освобождение выделенного диапазона портов.

Теперь можно собрать модуль, выполняя команду make, и проверить его установку и выгрузку с помощью команд sudo insmod и sudo rmmod, как это было сделано ранее. Также важно увидеть в списке сообщений ядра, что выводится сообщение о добавлении и удалении модуля с указанием старшего номера устройства, а также сообщение о выделении диапазона портов [0x100, 0x10b] или невозможности выполнить это выделение. Это можно сделать даже при отсутствии устройства ввода-вывода. При этом возможность или невозможность выделения указанного диапазона портов будет зависеть от конкретных особенностей конфигурации вычислительной системы (компьютера).

В случае рассматриваемого устройства ввода-вывода считывание цифрового кода, полученного в результате аналого-цифрового преобразования, реализуется следующим образом. Сначала в управляющий регистр записываются настройки параметров АЦП в виде одного байта данных. При этом 0-й бит управляющего регистра определяет автокомпенсацию нуля (для этого данный бит устанавливается в 0, иначе автомкомпенсация отключается). Но надо учитывать, что автокомпенсация в 2 раза увеличива-

ет время преобразования. 1-й бит управляющего регистра определяет автокалибровку (для этого он устанавливается в 0). В случае ввода аналогового значения этот бит должен быть равен 1. 2-й и 3-й биты задают коэффициент усиления для усилителя аналогового сигнала. Если оба этих бита равны 0, то коэффициент усиления равен 1. Если 2-й бит равен 1, а 3-й бит равен 0, то коэффициент усиления равен 10. Если 2-й бит равен 0, а 3-й бит равен 1, то коэффициент усиления равен 100. Оставшиеся четыре бита кодируют номер канала аналогового ввода, что соответствует номерам от 0 до 15.

Для записи байта в управляющий регистр достаточно записать этот байт в порт с адресом BASE_ADDRESS+4.

После записи байта в управляющий регистр выполняется запись какого-либо значения в порт с адресом BASE_ADDRESS. И эта запись является командой начала аналого-цифрового преобразования.

Признаком окончания этого процесса преобразования является установка в 1 значения 5-го бита в байте, соответствующем порту ввода-вывода с адресом BASE_ADDRESS+1.

По окончанию процесса преобразования можно прочитать полученный цифровой код. Первые 8 бит этого кода считываются из порта с адресом BASE_ADDRESS. Оставшиеся 4 бита соответствуют младшим 4-м битам порта с адресом BASE_ADDRESS+1. Также необходимо учесть знак (плюс или минус) цифрового кода. Значение этого знака кодируется значением 4-го бита в байте, соответствующем порту ввода-вывода с адресом BASE_ADDRESS+1.

Для получения цифрового кода из АЦП согласно изложенной процедуре в исходный текст драйвера после определения функции cleanup_module добавим определение функции getADC в виде следующего фрагмента

```
static unsigned int getADC(unsigned char channel,
```

```
                              unsigned char gain)
{
  unsigned int code;
  outb_p(2 | (gain << 2) | (channel << 4),
         BASE_ADDRESS+4);
  outb_p(0,BASE_ADDRESS);
  udelay(35);
  if (inb_p(BASE_ADDRESS+1) & 0x20) {
    code = inw_p(BASE_ADDRESS) & 0xfff;
    if(inb_p(BASE_ADDRESS+1) & 0x10)
      code = code - 4095;
  } else {
    code = 0xffff;
  }
  return code;
}
```

Эта функция возвращает цифровой код из АЦП.

В этой функции с помощью outb_p происходит запись байта в порт с заданным адресом. С помощью inb_p выполняется чтение байта из порта с заданным адресом. С помощью inw_p выполняется чтение двух байт из порта с заданным адресом.

Аргументы данной функции channel и gain определяют соответственно номер канала аналогового ввода и коэффициент усиления. Значения этих аргументов используются при формировании байта, который записывается в управляющий регистр. Кроме этого, в 0-й бит управляющего регистра записывается 0, что обеспечивает включение автокомпенсации нуля. Также в 1-й бит записывается 1, что отключает автокалибровку.

В случае включения автокомпенсации нуля преобразование длится примерно 30 микросекунд. При отключении автокомпенсации преобразование выполняется примерно в 2 раза быстрее. В данном примере автокомпенсация включена, поэтому перед проверкой окончания выполняется за-

держка в 35 микросекунд, что с запасом больше, чем предполагаемое время преобразования. Для реализации этой задержки используется функция udelay. Если после данной задержки ожидаемый бит не установлен в 1, то тогда цифровому коду присваивается значение 0xffff, что для простоты в дальнейшем может трактоваться в качестве сообщения об ошибке преобразования. Конечно, можно реализовать более удобные способы передачи подобного рода сообщений, но в данном примере используется этот простой способ. Если же преобразование успешно выполнено, то тогда формируется цифровой код с учетом знакового разряда.

Также в верхнюю часть файла надо добавить строку

```
#include <linux/delay.h>
```

Это необходимо для использования функции udelay, о которой было упомянуто выше.

Также для примера напишем функцию для вывода 12-ти разрядных цифровых кодов в ЦАП0 и ЦАП1 рассматриваемой платы ввода-вывода. Для вывода цифрового кода в ЦАП0 надо сначала записать его младшие 8 бит в порт с адресом BASE_ADDRESS+8 и оставшиеся 4 бита — в порт с адресом BASE_ADDRESS+9. Это можно сделать на основе двухбайтной записи в порт, выполняемой с помощью outw_p. В случае ЦАП1 все это выполняется аналогичным образом за исключением того, что используются порты с адресами BASE_ADDRESS+10, BASE_ADDRESS+11. После этого необходимо активировать вывод цифрового кода с помощью записи какого-либо значения в порт с адресом BASE_ADDRESS+6. Тогда фунция для вывода цифрового кода в ЦАП будет иметь вид

```
static void setDAC(unsigned char channel,
                   unsigned int code)
{
  if(channel==0){
    outw_p(code & 0xfff,BASE_ADDRESS+8);
  }else{
```

```
        outw_p(code & 0xfff,BASE_ADDRESS+10);
      }
      outb_p(0,BASE_ADDRESS+6);
    }
```

Эта функция имеет аргумент channel, определяющий целевой ЦАП (0 или 1), а также аргумент code, содержащий цифровой код, выводимый в данный ЦАП.

Теперь с помощью команды

```
sudo mkdir /dev/micropc5700
```

создадим каталог /dev/micropc5700.

Выполним команду

```
sudo mknod /dev/micropc5700/adc0 c N 0
```

где вместо N надо указать старший номер устройства (он выводится в списке сообщений ядра с помощью соответствующего сообщения при установке модуля драйвера). После успешного выполнения данной команды появляется файл /dev/micropc5700/adc0, и из которого можно читать. Этот файл соответствует 0-му каналу АЦП, то есть каналу AI_0 на структурной схеме платы ввода-вывода.

Выполним чтение из созданного устройства командой

```
cat /dev/micropc5700/adc0
```

и проверим список сообщения ядра. Там должна быть выведена строка

```
micropc5700: read
```

Аналогичным образом добавляем файлы для остальных 15-ти каналов с помощью команд вида

```
sudo mknod /dev/micropc5700/adcX c N X
```

где X принимает значения от 1 до 15. В результате должны сформироваться 16 файлов adc0, adc1, … adc15 в каталоге /dev/micropc5700. Здесь важно отметить, что параметр X определяет так называемый младший номер устройства, который может распознаваться в драйвере при открытии

файла. Этим можно воспользоваться, чтобы определять какой из этих 16-ти файлов открывается. Тем самым, определяется номер канала АЦП.

Похожим образом создадим два файла dac0 и dac1 для записи цифровых кодов в ЦАП0 и ЦАП1. Выполним команды

```
sudo mknod /dev/micropc5700/dac0 c N 16
sudo mknod /dev/micropc5700/dac1 c N 17
```

Тогда получится, что младший номер устройства, равный 16, соответствует файлу dac0 и ЦАП0, а младший номер устройства, равный 17, соответствует файлу dac1 и ЦАП1.

Также для разрешения записи в эти файлы выполним команды

```
sudo chmod a+w /dev/micropc5700/dac0
sudo chmod a+w /dev/micropc5700/dac1
```

Остается дополнить текст драйвера так, чтобы выполнялись правильные действия при тех или иных младших номерах устройства.

В верхней части файла разместим строку, создающую массив

```
static char msg[10];
```

Этот массив символов будет использоваться для вывода строки, содержащей информацию о цифровом коде АЦП.

Там же разместим строку

```
static char *msg_ptr;
```

Эта строка задает указатель, используемый в дальнейшем для перебора символов строки msg.

В функции device_open непосредственно перед строкой try_module_get(THIS_MODULE); разместим следующий фрагмент программы

```
int minor = MINOR(inode->i_rdev);
if (minor < 16){
  int code;
  if (emu_mode) {
    code = minor;
  } else {
```

```
    code = getADC(minor, 0);
  }
  sprintf(msg, "%d\n", code);
} else {
    msg[0] = 0;
}
msg_ptr = msg;
```

В ходе выполнения этого фрагмента происходит определение младшего номера устройства (minor) и формирование строки msg, содержащей информацию о цифровом коде АЦП. Если открывается файл с младшим номером устройства меньше 16, то предполагается, что это один из файлов вида adcX, где X равен значению minor. При этом формируется строка msg. В случае режима имитации ввода-вывода в эту строку записывается значение minor (младшего номера устройства). В случае работы с реальным устройством ввода-вывода в эту строку записывается цифровой код из АЦП, получаемый на основе вызова фунции getADC для канала ввода-вывода с номером minor и коэффициента усиления, равного 1 (что кодируется аргументом gain, равным 0). Если младший номер устройства не меньше 16, то предполагается, что открывается файл dac0 или dac1. В этом случае строка msg делается пустой. Также в этой функции указатель msg_ptr устанавливается на начало строки msg.

Теперь переходим к функции device_read. Ее содержимое заменяем на следующий фрагмент

```
if (*msg_ptr == 0)
  return 0;
int bytes_read = 0;
while (length && *msg_ptr) {
  put_user(*(msg_ptr++), buffer++);
  length--;
  bytes_read++;
}
```

```
return bytes_read;
```

Эта модифицированная функция теперь обеспечивает копирование строки msg по указателю buffer, использую функцию put_user. Строка msg ранее была сформирована при открытии файл с помощью функции device_open. После того, как строка msg полностью прочитана (скопирована в buffer), указатель msg_ptr указывает на значение 0, что приводит к возврату значения 0.

Также в верхнюю часть файла надо добавить строку

```
#include <asm/uaccess.h>
```

Это необходимо для использования функции put_user, которая вызывается из функции device_read.

Осталось изменить функцию device_write. Ее содержимое заменяем на следующий фрагмент

```
int minor = MINOR(filp->f_dentry->d_inode->i_rdev);
int code = 0;
sscanf(buff, "%d", &code);
int dac_id;
switch (minor) {
case 16:
  dac_id = 0; break;
case 17:
  dac_id = 1; break;
default:
  dac_id = -1;
}
if (dac_id == 0 || dac_id == 1) {
  if (emu_mode) {
    printk(KERN_INFO "%s: write %d to DAC%d\n",
           DEVICE_NAME, code, dac_id);
  } else {
    setDAC(dac_id, code);
  }
```

```
    }
    return len;
```

Эта модифицированная функция теперь обеспечивает определение младшего номера устройства (minor). После этого происходит формирование цифрового кода (code) на основе строки (buff), записываемой в файл. Отдельно можно еще проверять, насколько такая строка корректно сформирована, и выполнять необходимые действия при некорректной строке. Здесь для простоты эта проверка отсутствует. Затем выполняется определение номера ЦАП (dac_id) на основе значения minor. В случае режима имитации платы ввода-вывода выполняется вывод соответствующего сообщения с помощью функции printk. В случае использования реальной платы ввода-вывода (когда начальное значение переменной emu_mode устанавливается равной 0), происходит запись цифрового кода в ЦАП с номером dac_id с помощью ранее определенной функции setDAC.

Теперь можно собрать модуль, выполняя команду make, и проверить его установку и выгрузку с помощью команд sudo insmod и sudo rmmod, как это было сделано ранее. И можно проверить его работу выполняя команды вида

```
cat /dev/micropc5700/adcX
```

где вместо X можно задавать числа от 0 до 15, что соответствует каналам аналогового ввода от 0 до 15. В случае режима имитации платы ввода-вывода (если начальное значение переменной emu_mode установлено равным 1) в консоль будет выводиться значение X. Так например, в ответ на команду

```
cat /dev/micropc5700/adc10
```

в консоль будет выведено число 10.

При наличии работе с реальной платой ввода-вывода надо задать значение переменной emu_mode, равным 0, и собрать драйвер заново. После этого, в ответ на указанные команды в консоль должен выводится циф-

ровой код, соответствующий значению аналогового сигнала, который поступает на канал аналогового ввода с номером X.

Аналогично можно проверить, как драйвер реагирует на запись значений в файлы /dev/micropc5700/dac0 и /dev/micropc5700/dac1. Например, можно выполнить команду

```
echo 128 > /dev/micropc5700/dac1
```

Если используется режим имитации платы ввода-вывода, то тогда в список сообщений ядра (который вызывается командой dmesg) добавляется строка вида

```
micropc5700: write 128 to DAC1
```

При работе с реальной платой ввода-вывода в результате выполнения данной команды должен быть осуществлен вывод цифрового кода 128 в ЦАП0.

С помощью данного простого примера драйвера для устройства ввода-вывода показаны некоторые принципы разработки подобных драйверов. Более подробно вопросы, связанные с разработкой драйверов устройств, рассматриваются в ходе соответствующих практических занятий и лабораторных работ.

2.4. Контрольные вопросы

1. Каковы особенности интегрированных сред разработки и работы в этих средах?
2. Каковы особенности отладки программы в интегрированной среде разработки?
3. Какие существуют подходы, призванные упростить практическое внедрение моделей и методов планирования задач реального времени?
4. Может ли использоваться ядро Linux для создания управляющей СРВ?

5. Каковы основные функции инструментальных программных средств для поддержки инженерных решений применительно к планированию задач реального времени?
6. Каковы типовые варианты использования системного программного обеспечения при разработке управляющих систем реального времени?
7. Что такое кросс-разработка ПО для управляющих СРВ?
8. Для чего может понадобится использовать потоки при разработке ПО управляющей СРВ?
9. Какие проблемы могут возникнуть при выполнении нескольких потоков?
10. Какие проблемы может решить использование мьютексов при использовании нескольких потоков?
11. Каковы особенности разработки драйверов применительно к ядру Linux?

Заключение

В учебном пособии изложены базовые понятия, относящиеся к: системному программному обеспечению; управляющим системам реального времени; планированию задач реального времени. Также рассмотрены: основы работы с интегрированной средой разработки Code::Blocks; пример разработки программ, использующих потоки POSIX; пример разработки прототипа драйвера применительно к ядру Linux. Пособие является учебным материалом для дисциплины «Системное программное обеспечение управляющих систем реального времени», которая является частью сетевой магистерской программы 22040056.68 «Информационные технологии в проектировании управляющих систем реального времени» по направлению 220400 «Управление в технических системах».

Главным компонентом в моделях управляющих систем реального времени являются задачи реального времени, реализация которых во многом зависит от особенностей выбранного системного программного обеспечения. Поэтому понятие задачи реального времени подробно рассмотрено в пособии. Аспекты реального времени привносятся в модель, в первую очередь, за счет формализации ограничений реального времени, которые затем учитываются в процессе планирования задач реального времени. Базовым моделям планирования задач реального времени посвящен существенный объем материала данного пособия.

Важной областью деятельности магистрантов является научно-исследовательская деятельность. Поэтому в пособии приведен обзор современного состояния исследований, связанных с планированием задач реального времени систем управления. Это представляется важным, так как на русском языке очень мало имеется литературы, посвященной подобным исследованиям.

К пособию приложен подробный библиографический список, вклю-

чающий важные научные и обзорные работы, а также монографии, посвященные проблемам реального времени, связанным с системным программным обеспечением управляющих систем реального времени.

Библиографический список

1. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982. – 416 с.

2. Дейтел Х.М, Дейтел П.Д. Как программировать на С: пер. с англ. М.: Бином-Пресс, 2009.— 910 с.

3. Керниган Б., Ритчи Д. Язык программирования С: пер. с англ. М.: Вильямс, 2009 .— 304 с.

4. Ключев А.О., Кустарев П.В., Ковязина Д.Р., Петров Е.В. Программное обеспечение встроенных вычислительные систем. – СПб.: СПбГУ ИТМО, 2009. – 212 с. URL: http://window.edu.ru/resource/411/63411/files/itmo368.pdf (дата обращения 16.09.2013).

5. Подбельский В.В., Фомин С.С. Курс программирования на языке Си. – М.: ДМК Пресс, 2003.— 384 с.: ил.

6. Разработка приложений для встраиваемых устройств: Часть 2. Применение Code::Blocks для разработки AVR-приложений [Электронный ресурс] // IBM developerWorks: Ресурс IBM для разработчиков и IT профессионалов. URL: http://www.ibm.com/developerworks/ru/library/l-CodeBlocks_and_avr-gcc/index.html (дата обращения 16.09.2013).

7. Разработка приложений для встраиваемых устройств: Часть 4. Применение Code::Blocks для разработки SDCC-приложений [Электронный ресурс] // IBM developerWorks: Ресурс IBM для разработчиков и IT профессионалов. URL: http://www.ibm.com/developerworks/ru/library/l-codeblocks_and_sdcc/index.html (дата обращения 16.09.2013).

8. Сушков Б.Г. ЭВМ управляет экспериментом (вычислительные системы реального времени). – М.: Знание, 1987. – 32 с.

9. Танаев В.С., Гордон В.С., Шафранский Я.М. Теория расписаний. Одностадийные системы. – М.: Наука, 1984. – 381 с.

10. Теория расписаний и вычислительные машины / Под ред. Коффмана Э.Г. – М.: Наука, 1984. – 334 с.

11. Шилдт Г. Полный справочник по C: Пер. с англ. – М.: Вильямс, 2009 .— 704 с.

12. Abeni L., Buttazzo G. Integrating Multimedia Applications in Hard Real-Time Systems // Proceedings of 19th IEEE Real-Time Systems Symposium. – Madrid, Spain, 1998. – P. 4–13.

13. Anta A., Tabuada P. On the benefits of relaxing the periodicity assumption for control tasks // Proceedings Real-Time Embedded Technol. Appl. Symp.(RTAS). Work-in-Progress Track. 2008. – P. 57–60.

14. Anta A., Tabuada P. On the benefits of relaxing the periodicity assumption for networked control systems over CAN // 30th IEEE Real-Time Systems Symposium, 2009. – P. 3–12.

15. Audsley N.C., Burns A., Richardson M.F., Wellings A. J. Incorporating Unbounded Algorithms Into Predictable Real-Time Systems. Department of Computer Science, University of York, UK September 1991.

16. Audsley N.C. Deadline Monotonic Scheduling, September 1990, Technical Report YCS146, Department of Computer Science, University of York. – 38 p.

17. Audsley N.C. Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Technical Report YCS164, Department of Computer Science, University of York, UK, 1991. – 31 p.

18. Audsley N., Tindell K., Burns A. The End of the Line for Static Cyclic Scheduling. Technical Report, University of York, England, 1993.

19. Baruah S., Chen D., Gorinsky S., Mok A. Generalized multiframe tasks // Real-Time Systems, Vol. 17, No. 1, 1999. – P. 5–22.

20. Baruah S. Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks // Real-Time Sys-tems, Vol. 24, No. 1, 2003. – P. 93–128.

21. Baruah S. The Non-cyclic Recurring Real-Time Task Model // Real-Time Systems Symposium (RTSS), 2010. – P. 173–182.

22. Bate I., Burns A. A Framework for Scheduling in Safety-Critical Embedded Control Systems // Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), 1999.

23. Ben Gaid M.E.M., Cela A.S., Hamam Y. Optimal real-time scheduling of control tasks with state feed-back resource allocation // IEEE Transactions on Control Systems Technology, Vol. 17, No. 2, 2009. – P. 309–326.

24. Bernat G., Burns A. Liamosi A. Weakly hard real-time systems // IEEE Transactions on Computers, Vol. 50, No. 4, 2001. – P. 308–321.

25. Bernat G. Specification and Analysis of Weakly Hard Real-Time Systems, PhD thesis, 1998.

26. Burns A., Bernat G. Jorvik: A Framework for Effective Scheduling. Department of Computer Science, University of York, York, 2001.

27. Burns A. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. Technical Report YCS-93-214, University of York, 1993. – 19 p.

28. Buttazzo G. Hard Real-Time Computing Systems. – Springer, 2011. – 521 p.

29. Caccamo M., Lipari G., Buttazzo G. Sharing Resources among Periodic and Aperiodic Tasks with Dynamic Deadlines // Proceedings of the IEEE Real-Time Systems Symposium, Phoenix, Arizona, December 1999.

30. Chantem T., Hu X.S., Lemmon M.D. Generalized elastic scheduling for real-time tasks // IEEE Transactions on Computers, Vol. 58, No. 4, 2009. – P. 480–495.

31. Choi B.W., Shin D.G., Park J.H., Yi S.Y., Gerald S. Real-time control architecture using Xenomai for intelligent service robots in USN environments // Intelligent Service Robotics, Vol. 2, No. 3, 2009. – P. 139–151.

32. chronSIM. Real-time simulator [Электронный ресурс] // INCHRON. URL: http://www.inchron.com/chronsim.html (дата обращения 16.09.2013).

33. chronVAL. Real-Time Validator [Электронный ресурс] // INCHRON. URL: http://www.inchron.com/chronval.html (дата обращения 16.09.2013).

34. Code::Blocks [Электронный ресурс]. URL: http://www.codeblocks.org (дата обращения 16.09.2013).

35. Davis R.I. Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems // Technical Report YCS-93-216 / University of York. – York, 1993. – 28 p.

36. Davis R.I., Burns A. Hierarchical Fixed Priority Pre-emptive Scheduling. Technical Report YCS-2005-385, Department of Computer Science, University of York, UK, 2005. – 58 p.

37. Davis R.I. On Exploiting Spare Capacity in Hard Real-Time Systems, PhD Thesis, University of York, 1995. – 283 p.

38. Davis R.I., Tindell K.W., Burns A. Scheduling slack time in fixed priority pre-emptive systems // Real-Time Systems Symposium, 1993. – P. 222–231.

39. Dobrin R., Fohler G., Puschner P. Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling // Proceedings of 22nd IEEE Real-Time Systems Symposium. – London, 2001. – P. 225–234.

40. Dozio L., Mantegazza P. Real-time distributed control systems using RTAI // Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003. – P. 11–18.

41. ERIKA Enterprise and RT-Druid Features [Электронный ресурс] // Erika Enterprise and RT-Druid. URL: http://erika.tuxfamily.org/drupal/ (дата обращения 16.09.2013).

42. Faggioli D., Checconi F., Trimarchi M., Scordino C. An EDF scheduling class for the Linux kernel // Proc. of the Real-Time Linux Workshop. 2009.

43. Fersman E., Krcal P., Pettersson P., Yi W. Task automata: Schedulability, decidability and undecidability // Information and Computation, Vol. 205, No. 8, 2007. – P. 1149–1172.

44. Fohler G. Dynamic Timing Constraints – Relaxing Over-constraining Specifications of Real-Time Systems // Proceedings of Work-in-Progress Session, 18th IEEE Real-Time Systems Symposium, 1997. – P. 27–30.

45. Fohler G., Flexibility in Statically Scheduled Hard Real-Time Systems. Dissertation, Eingereicht an der Technischen Universitat Wien, Technisch-Naturwissenschaftliche Fakultat, Wien, 1994. – 101 p.

46. Gardner M.K., Liu J.W.S. Performance of algorithms for scheduling real-time systems with overrun and overload // 11th Euromicro Conference on Real-Time Systems. June 1999.

47. GCC, the GNU Compiler Collection [Электронный ресурс]. URL: http://gcc.gnu.org (дата обращения 16.09.2013).

48. GNU GPL 3 человеческим языком [Электронный ресурс] URL: http://licenseit.ru/wiki/index.php/Статья:GNU_GPL_3_человеческим_языком (дата обращения 16.09.2013).

49. Guardia J., and Marti P., Velasco M., Castane R.. Enabling Feedback Scheduling in TrueTime // Re-search Report ESAII-RR-06-05, Automatic Control Department, Technical University of Catalonia, 2006. – 20p.

50. Guerra R., Fohler G. A gravitational task model with arbitrary anchor points for target sensitive real-time applications // Real-Time Systems, Vol. 43, No. 1, 2009. – P. 93–115.

51. Gutierrez J. P., Harbour M. G. Schedulability Analysis for Tasks with Static and Dynamic Offsets // Proceedings 19th IEEE Real-Time Systems Symposium, 1998.

52. Isovic D., Fohler G. Handling mixed sets of tasks in combined offline and online scheduled real-time systems // Real-Time Systems, Vol. 43, No. 3, 2009. – P. 296–325.

53. Kopetz H. Real-time systems: design principles for distributed embedded applications. Springer, 2011.

54. Kopetz H., Zainlinger R., Fohler G., Kantz H., Puschner P., Schutz W. An Engineering Approach to Hard Real-Time System Design // Institut fur Technische Informatik Technische Universitat Wien Treitlstr. 3/182, Austria, 1998.

55. Lehoczky J. P., Ramos-Thuel S. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks Fixed Priority Pre-emptive systems // Proceedings IEEE Real-Time Systems Symposium, 1992. – P. 110-123.

56. Lehoczky J. P., Sha L., Strosnider J. K. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments // Proceedings IEEE Real-Time System Symposium, San Jose, 1987. – P. 261-270.

57. Liu C. L., Layland J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment // Journal of the Association for Computing Machinery, Vol. 20, No. 1, January 1973. – P. 46-61.

58. Liu J.,Lin K., Shih W., Yu A., Chung J., Zhao W. Algorithms for Scheduling Imprecise Computations // IEEE Computer, May 1991.

59. Mäki-Turja J., Nolin M. Fast and Tight Response-Times for Tasks with Offsets // Proceedings of 17th Euromicro Conference on Real-Time Systems. 2005.

60. Mäki-Turja J., Nolin M. Faster Response Time Analysis of Tasks With Offsets // Proceedings 10th IEEE Real-Time Technology and Applications Symposium, 2004.

61. Marinoni M., Facchinetti T., Buttazzo G., Franchino G. An embedded real-time system for autonomous flight control // Proc ANIPLA International Congress on Methodologies for Emerging Technologies in Automation. 2006.

62. MaRTE OS Home Page [Электронный ресурс] // MaRTE OS. Minimal Real-Time OS. URL: http://marte.unican.es (дата обращения 16.09.2013).

63. Marti P., Fuertes J.M., Fohler G., Ramamritham K. Jitter compensation for real-time control systems // 22nd IEEE Real-Time Systems Symposium, 2001. – P. 39–48.

64. MinGW – Minimalist GNU for Windows [Электронный ресурс]. URL: http://www.mingw.org (дата обращения 16.09.2013).

65. Mok A.K., Chen D. A multiframe model for real-time tasks // IEEE Transactions on Software Engineering, Vol. 23, No. 10, 1997. – P. 635–645.

66. Mok A.K. Fundamental design problems of distributed systems for the hard-real-time environment // PhD Thesis, MIT, Cambridge, Massachusetts, USA, 1983. – 182 p.

67. Mossige M., Sampath P., Rao R.G. Evaluation of Linux rt-preempt for embedded industrial devices for Automation and Power Technologies - A Case Study // 9th RTL Workshop, 2007.

68. Moyo N.T., Nicollet E., Lafaye F., Moy C. On schedulability analysis of non-cyclic generalized multi-frame tasks // 22nd Euromicro Conference on Real-Time Systems (ECRTS), 2010. – P. 271–278.

69. Olderog E.-R., Dierks H. Real-Time Systems. Formal Specification and Automatic Verification. – Cambridge University Press. 2008. – 320 p.

70. Peng D., Zhang H., Huang C., Lin J., Li H. Study of Immune PID Networked Control System Based on TrueTime // Journal of Networks, Vol. 6, No 6, 2011. – P. 912–915.

71. Ramamritham K., Stankovic J. Scheduling Algorithms and Operating Systems Support for Real-Time Systems // Proceedings of the IEEE, Vol. 82, No. 1, January 1994. – P. 55-67.

72. Romeo. A tool for Time Petri Nets analysis [Электронный ресурс] // Real-Time Systems group. URL: http://romeo.rts-software.org/?page_id=2 (дата обращения 16.09.2013).

73. RTAI - the RealTime Application Interface for Linux from DIAPM [Электронный ресурс] // RTAI - Real Time Application Interface Official Website. URL: https://www.rtai.org (дата обращения 16.09.2013).

74. RTSIM. Real-Time system SIMulator [Электронный ресурс] // RTSIM Home Page. URL: http://rtsim.sssup.it (дата обращения 16.09.2013).

75. S.Ha.R.K.: Soft Hard Real-Time Kernel [Электронный ресурс] // The S.Ha.R.K. project. URL: http://shark.sssup.it (дата обращения 16.09.2013).

76. Sandström K., Norström C. Managing Complex Temporal Requirements in Real-Time Control Systems // Proceedings of 9th IEEE Conference on Engineering of Computer-Based Systems. – Lund, 2002. – P. 103-109.

77. Sha L., Abdelzaher T., Årzén K. E., Cervin A., Baker T., Burns A., Buttazzo G., Caccamo M., Lehoczky J., Mok A. K. Real-Time Scheduling Theory: A Historical Perspective // Real-Time Systems 28, 2004. – P. 101–155.

78. Sha L., Lehoczky J. P., Rajkumar R. Solutions For Some Practical Problems in Prioritised Preemptive Scheduling // Proceedings IEEE Real-Time Systems Symposium, 1986. – P. 181-191.

79. Sha L., Sprunt B., Lehoczky J. P. Aperiodic Task Scheduling for Hard Real-Time Systems // Real-Time Systems 1(1), 1989. – P. 27-69.

80. Singhoff F., Plantec A. AADL modeling and analysis of hierarchical schedulers // ACM SIGAda Ada Letters, Vol. 27, No. 3, 2007. – P. 41–50.

81. Sprunt B., Lehoczky J., Sha L. Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm // Proceedings of IEEE Real-Time Systems Symposium, December 1988. – P. 251-258.

82. Spuri M., Buttazzo G. Scheduling Aperiodic Tasks in Dynamic Priority Systems // Real-Time Systems, vol. 10, 1996. – P. 179-210.

83. Stankovic J. Real-Time Computing // BYTE, invited paper, August 1992. – P. 155-160.

84. Stankovic J., Spuri M., Di Natale M., Buttazzo G. Implications of Classical Scheduling Results for Real-Time Systems // IEEE Computer, Vol. 28, No. 6, June 1995. – P. 16-25.

85. Stigge M., Ekberg P., Guan N., Yi W. The digraph real-time task model // 17th Real-Time and Embed-ded Technology and Applications Symposium (RTAS), 2011. – P. 71–80.

86. STORM. Simulation TOol for Real time Multiprocessor scheduling [Электронный ресурс] // Real-Time Systems group. URL: http://storm.rts-software.org/doku.php (дата обращения 16.09.2013).

87. Sun J., Liu J. W. S. Bounding completion times of jobs with arbitrary release times and variable execution times //Proceedings of the IEEE Real-Time Systems Symposium, Washington D.C., 1996. – P. 164-173.

88. SymTA/S [Электронный ресурс] // Symtavision. URL: http://symtavision.com/symtas.html (дата об-ращения 12.03.2012).

89. The Cheddar project: a free real time scheduling analyzer // Université de Bretagne Occidentale. URL: http://beru.univ-brest.fr/~singhoff/cheddar (дата обращения 16.09.2013).

90. Tia T.-S., Deng Z., Shankar M., Storch M., Sun J., Wu L.-C., Liu J.W.-S. Probabilistic performance guarantee for real-time tasks with varying computation times // Proceedings, Real-Time Technology and Applications Symposium, Chicago, Illinois, May 1995. – P. 164-173.

91. Tia T.-S., Liu J.W.S., Shankar M. Aperiodic Request Scheduling in Fixed-Priority Preemptive Systems. Technical Report UIUCDCS-R-94-1859, University of Illinois at Urbana-Champaign, 1994.

92. Tia T.-S. Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-Time Systems. PhD thesis, University of Illinois at Urbana-Champaign, April 1995. – 104 p.

93. Tindell K. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. Technical Report YCS-92-189, University of York, 1992. – 16 p.

94. Tindell K., Burns A., Wellings A. Allocating Hard Real-Time Tasks (An NP-Hard Problem Made Easy). Technical Report, University of York, 1995.

95. Tindell K. Using Offset Information to Analyse Static Priority Pre-emptively Scheduled Task Sets. Technical Report YCS-182, Dept. of Computer Science, University of York, England, 1992. – 17 p.

96. TrueTime: Simulation of Networked and Embedded Control Systems [Электронный ресурс] // Department of Automatic Control, Lund University. URL: http://www3.control.lth.se/truetime (дата обращения 16.09.2013).

97. Velasco M., Marti P., Bini E. Control-driven tasks: Modeling and analysis // Real-Time Systems Symposium, 2008. – P. 280–290.

98. Wan J., Li D. Fuzzy feedback scheduling algorithm based on output jitter in resource-constrained embedded systems // 2010 International Conference on Challenges in Environmental Science and Computer Engineering (CESCE), Vol. 2, 2010. – P. 457–460.

99. Wang W., Mok A.K., Fohler G. Generalized Pre-Scheduler // Proceedings of 16th Euromicro Conference on Real-Time Systems. – Catania, 2004. – P. 127-134.

100. Wang X., Lemmon M. Event-triggering in distributed networked systems with data dropouts and delays // Hybrid systems: Computation and control, 2009. – P. 366–380.

101. What Is RTLinux? [Электронный ресурс] // RTLinuxFree. URL: http://www.rtlinuxfree.com/license.html (дата обращения 16.09.2013).

102. Williams R. Real-Time Systems Development. – Elsevier. 2006. – 455 p.

103. Xenomai: Real-Time Framework for Linux Электронный ресурс] // Xenomai. URL: http://www.xenomai.org/index.php/Main_Page (дата обращения 16.09.2013).

104. Xia F., Sun Y. Control and Scheduling Codesign. – Springer. 2008. – 246 p.

105. Zhuang S., Cassandras C.G. Optimal control of discrete event systems with weakly hard real-time constraints // Discrete Event Dynamic Systems, Vol. 19, No. 1, 2009. – P. 67–89.