ЮРИЙ МАГДА

bhv®

ПРОГРАММИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ИНТЕРФЕЙСОВ

ПОСЛЕДОВАТЕЛЬНЫЕ ИНТЕРФЕЙСЫ: ПРОТОКОЛЫ И СИГНАЛЫ

ПРОГРАММИРОВАНИЕ АСИНХРОННЫХ ПРИЕМОПЕРЕДАТЧИКОВ

ПРОГРАММИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ИНТЕРФЕЙСОВ В WINDOWS И LINUX

ПОСЛЕДОВАТЕЛЬНЫЕ ИНТЕРФЕЙСЫ В СЕТИ ИНТЕРНЕТ: ПРАКТИКА ПРОГРАММИРОВАНИЯ

ВИРТУАЛЬНЫЕ ПОСЛЕДОВАТЕЛЬНЫЕ ПОРТЫ В РАЗРАБОТКЕ И ОТЛАДКЕ ПРИЛОЖЕНИЙ

Юрий Магда

ПРОГРАММИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ИНТЕРФЕЙСОВ

Санкт-Петербург
«БХВ-Петербург»
2009

УДК 681.3.068
ББК 32.973.26-018.1
М12

Магда Ю. С.

М12 Программирование последовательных интерфейсов. — СПб.: БХВ-Петербург, 2009. — 304 с.: ил. + CD-ROM — (Профессиональное программирование)

ISBN 978-5-9775-0274-0

Рассматривается широкий круг вопросов функционирования последовательных интерфейсов обмена данными. Проанализированы основные протоколы последовательного обмена данными, характеристики сигналов и базовые аппаратные средства на основе асинхронных приемопередатчиков. Подробно изложена методика программирования протоколов последовательного обмена на низком уровне. Значительная часть материала книги посвящена программированию последовательного обмена данными в популярных операционных системах Windows и Linux, а также разработке приложений для Интернета. Рассмотрены методы разработки программного обеспечения с использованием виртуальных последовательных интерфейсов обмена данными. Прилагаемый компакт-диск содержит файлы с исходными текстами описанных в книге программ.

Для программистов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Рожко*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.09.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 24,51.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0274-0

Оглавление

Введение

Последовательный интерфейс является одним из наиболее ранних интерфейсов обмена данными, но, несмотря на это, он до сих пор является одним из самых распространенных. Это обусловлено его относительной простотой в реализации, надежностью, доступностью оборудования. Существенно и то, что программирование последовательного обмена данными довольно легко выполняется с помощью стандартных функций прикладного интерфейса программирования, которые доступны во всех популярных операционных системах, таких как Windows и Linux.

Последовательный интерфейс очень широко используется как в простейших коммуникационных системах, так и в большинстве промышленных систем обмена данными, сбора информации и управления, поскольку при создании надежных распределенных систем управления и контроля стандартам последовательного обмена данными RS-232 и RS-485 реальной альтернативы пока нет.

Основной недостаток последовательного интерфейса — относительно низкая скорость обмена данными по сравнению с другими интерфейсами — в настоящее время успешно преодолевается в новых аппаратных разработках, а аппаратно-программные методы "виртуализации" позволяют работать практически с любым количеством последовательных интерфейсов в одной системе.

Интерес к последовательному интерфейсу в последнее время не только не ослабевает, но даже возрастает, о чем свидетельствует как огромное количество выпускаемого оборудования (промышленного и лабораторного), работающего с последовательным интерфейсом, так и большое количество статей и книг, выпускаемых за рубежом по данной тематике.

В книге рассматриваются вопросы аппаратной архитектуры и программирования последовательного интерфейса в операционных системах Windows и Linux. Детально анализируются аппаратно-архитектурные особенности последовательного интерфейса персонального компьютера, стандарты и интерфейсы, методы тестирования и настройки. Большое внимание уделено анализу практических проектов аппаратно-программных систем обмена данными с использованием интерфейсов RS-232 и RS-485. Все примеры программного кода, приведенные в книге, имеются на компакт-диске.

Книга существенно отличается от аналогичных изданий. Во-первых, подробно рассмотрен аппаратный интерфейс протоколов последовательной передачи данных на базе универсальных асинхронных приемопередатчиков (*UART* — Universal Asynchronous Receiver/Transmitter) и проанализированы вопросы программирования этих устройств в защищенных операционных системах Windows. Во-вторых, на большом числе практических примеров рассмотрены методы программирования последовательного порта в наиболее популярных средах программирования, таких как Delphi и Microsoft Visual Studio .NET последних версий. Программирование последовательного порта в среде .NET и Delphi, благодаря введению новых компонентных моделей для данного интерфейса, стало значительно более эффективным и расширило возможности разработчиков систем обработки и обмена данными. В-третьих, материал книги раскрывает аспекты создания и программирования сетевых систем сбора и обработки данных, на нижнем уровне которых функционирует последовательный интерфейс.

Книга состоит из шести глав, краткое описание каждой из них приведено далее.

- *Глава 1. "Последовательный интерфейс в системах обмена данными".*

  Материал главы посвящен обзору возможностей протоколов последовательного обмена данными и областей их применения. Здесь приводятся и общие характеристики наиболее распространенных стандартов последовательного обмена данными.

- *Глава 2. "Стандарты и протоколы последовательного интерфейса".*

  В главе подробно рассматриваются характеристики популярных протоколов обмена данными RS-232 и RS-485, их аппаратная реализация и программирование на низком уровне. Значительная часть материала посвящена практическим аспектам реализации аппаратных интерфейсов и их программированию на примерах простых систем последовательной передачи данных.

- *Глава 3. "Аппаратная реализация последовательного интерфейса в компьютерных системах".*

  Здесь рассмотрены вопросы аппаратной реализации интерфейса RS-232 в персональных компьютерах. На многочисленных примерах демонстрируются методы программирования стандартных асинхронных приемопередатчиков на низком уровне в операционных системах Windows.

- *Глава 4. "Программирование последовательного интерфейса в операционных системах Windows".*

  В этой главе анализируются основные аспекты программирования последовательного обмена данными с использованием функций интерфейса прикладного программирования операционных систем Windows. Значительная часть материала главы посвящена программированию приложений с графическим интерфейсом пользователя в Microsoft Visual Studio .NET и Delphi.

- *Глава 5. "Программирование последовательного интерфейса в Linux".*

  Материал этой главы посвящен программированию последовательного обмена данными в операционных системах Linux. Приводятся многочисленные примеры программного кода на языке GNU C.

- *Глава 6. "Расширения ввода-вывода последовательного интерфейса".*

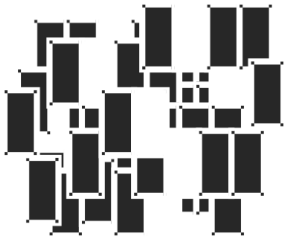
  В главе рассматриваются современные аппаратно-программные технологии программирования приложений для работы с последовательными интерфейсами на основе преобразования RS-232 в *USB* (Universal Serial Bus), а также использование виртуальных последовательных портов. Значительная часть главы посвящена разработке простых систем обработки данных с использованием интернет-технологий с детальным анализом программного кода.

Материал книги будет полезен всем желающим, которые хотели бы существенно пополнить свои знания в области разработки и программирования систем передачи данных на основе последовательных интерфейсов.

Благодарности

Автор выражает огромную благодарность сотрудникам издательства "БХВ-Петербург" за подготовку материалов книги к изданию. Особая признательность жене Юлии за понимание и поддержку при подготовке рукописи.

Глава 1



Последовательный интерфейс в системах обмена данными

Обмен данными между различными устройствами телекоммуникационных и компьютерных систем принципиально можно выполнить методом параллельной или последовательной передачи всех битов данных. При последовательной передаче данные в коммуникационный канал передаются бит за битом, синхронно с сигналами тактовой частоты, которая определяет скорость обмена данными. Последовательная передача данных используется при обмене данными между удаленными системами (компьютерными или телекоммуникационными) и обеспечивает надежную передачу данных на большие расстояния. Для последовательной передачи данных разработаны и очень широко применяются стандарты RS-232, RS-485, SPI, I2C, 1-Wire, которые используются, как правило, в компьютерных системах общего назначения. На базе протоколов RS-232 и RS-485 разработаны и специализированные интерфейсы для обмена данными в промышленных сетях. Наибольшее распространение в таких сетях получили протоколы Profibus, Modbus, Fieldbus и др.

При сравнении систем параллельной передачи данных, в которых все биты данных передаются одновременно, с последовательными системами может показаться, что параллельная передача данных всегда будет выполняться быстрее. Теоретически это верно, но на практике не все так однозначно. Во многих случаях при обмене данными между удаленными системами линии последовательной передачи данных синхронизируются на более высокой частоте, чем это делается в системах параллельной передачи данных, что обеспечивает и более высокую скорость передачи.

Преимуществом систем последовательной передачи данных по сравнению с системами параллельной передачи является то, что в них используется на-

много меньше сигнальных линий, а это упрощает реализацию физического интерфейса и значительно удешевляет стоимость оборудования. Физически последовательный обмен данными можно реализовать намного меньшим количеством сигнальных линий (при параллельной передаче данных потребуется как минимум 8 линий данных плюс сигнал строба, при этом мы не учитываем сигналы квитирования как со стороны передатчика, так и со стороны приемника). Иногда при параллельной передаче пытаются уменьшить количество сигнальных линий, используя две 4-битовые посылки данных при передаче одного байта, но при такой "гибридной" конфигурации возникают дополнительные проблемы.

Хочу отметить, что все внутренние шины обмена данными в самом оборудовании, например, в системах на базе процессоров и микроконтроллеров, реализованы по принципу параллельной передачи, поскольку при небольших внутрисхемных расстояниях параллельная шина при прочих равных условиях будет всегда работать быстрее. Тем не менее, для удаленных систем в настоящее время наиболее оптимальным является обмен данными по последовательным линиям.

Рассмотрим вкратце наиболее распространенные стандарты (интерфейс) последовательной передачи данных и области их применения. Начнем с RS-232.

Интерфейс *RS-232* используется, как правило, для подключения к компьютеру стандартных внешних устройств (принтера, сканера, модема, мыши и др.), а также для связи компьютеров между собой на относительно небольших расстояниях. Широкое применение интерфейс RS-232 находит при настройке и диагностике различного телекоммуникационного и компьютерного оборудования. Все современные системы выпускаются с внешним интерфейсом последовательного порта и прошитым программным обеспечением, которое позволяет настраивать такое оборудование с помощью простых терминальных программ, запущенных, например, на персональном компьютере.

Если сравнивать, например, интерфейс RS-232 и интерфейс параллельного порта *SPP* (Standard Parallel Port) (Centronics) персонального компьютера, то при использовании первого данные можно передавать на значительно бóльшие расстояния, при этом соединительный кабель можно реализовать в трехпроводной конфигурации. Программирование последовательного интерфейса, с другой стороны, несколько сложнее, чем параллельного порта, особенно если выполнять его на уровне аппаратных средств системы. Тем не менее, нужно отметить, что все компьютерные системы общего назначения работают, как правило, под управлением одной из популярных операцион-

ных систем (чаще всего Windows), в которых предусмотрен специальный программный интерфейс для работы с файлами и периферийными устройствами, значительно упрощающий работу с последовательным портом на программном уровне. Например, в операционных системах Windows для работы с последовательным портом можно использовать функции прикладного интерфейса программирования *WIN API* (Windows Application Programming Interface).

Передача данных по интерфейсу RS-232 осуществляется побайтово, причем начало и конец передачи синхронизируются стартовым и стоповым битами. Сам процесс передачи данных можно контролировать как программными, так и аппаратными средствами (это зависит от конфигурации системы). Данные между устройствами могут передаваться как в одном (*полудуплексный режим*), так и в обоих направлениях (*дуплексный режим*). Физическая линия подсоединяется к интерфейсу RS-232 посредством стандартного 25-контактного разъема DB25 (в настоящее время используется редко) или 9-контактного разъема DB9.

Интерфейс RS-232 может соединять только два устройства (конфигурация "точка-точка"), при этом линия передачи одного устройства соединяется с линией приема другого и наоборот. Эта конфигурация используется при полнодуплексном режиме. При работе в полудуплексном режиме передающая линия одного устройства соединяется с приемной линией другого. Различные сигналы квитирования (подтверждения) могут генерироваться как аппаратными средствами (за счет использования дополнительных линий управления), так и программным способом (например, включением в поток передаваемых данных соответствующих управляющих символов).

Вот основные характеристики интерфейса RS-232:

- скорость передачи — до 115 Кбит/с;
- расстояние передачи — до 15 м;
- тип сигнала — несимметричный по напряжению, один передатчик и один приемник;
- тип соединения — полный дуплекс, одноточечное соединение.

Попытки увеличить расстояние, на которое могут передаваться данные, и скорость передачи данных привели к модернизации интерфейса RS-232. В результате появился интерфейс *RS-423*, который мог обеспечивать высокоскоростную передачу данных (до 10 Мбит/с) на расстояние до 1200 м в несимметричной конфигурации, а также ставший наиболее популярным в про-

мышленных системах интерфейс *RS-485*, который обеспечивает передачу симметричного (дифференциального) сигнала.

Интерфейс RS-485 реализует двунаправленную полудуплексную передачу данных и допускает одновременное подключение 32 приемников к шине. На базе интерфейса RS-485 созданы и получили широкое распространение многие промышленные стандарты обмена данными, при этом RS-485 обеспечивает канальный уровень (если рассматривать промышленные шины в виде стека протоколов).

Рассмотрим основные технические характеристики интерфейса RS-485 более подробно. Он является наиболее широко используемым промышленным стандартом, использующим двунаправленную сбалансированную линию передачи. Протокол поддерживает многоточечные соединения, обеспечивая создание сетей с количеством узлов до 32 и передачу данных на расстояние до 1200 м. Использование буферных усилителей сигнала в интерфейсе RS-485 позволяет увеличить расстояние передачи еще на 1200 м или добавить еще 32 узла. Стандарт RS-485 поддерживает полудуплексный режим обмена данными, при этом для передачи и приема данных достаточно одной витой пары проводников.

Интерфейс RS-485 имеет следующие характеристики:

- скорость передачи — до 10 Мбит/с;
- расстояние передачи — до 1200 м;
- тип сигнала — дифференциальное напряжение;
- тип линии передачи — витая пара, до 32 передатчиков и до 32 приемников;
- тип соединения — полудуплекс, многоточечное соединение.

На основе протоколов RS-232, RS-422 и RS-485 разработаны практически все популярные промышленные стандарты. Рассмотрим некоторые из них и начнем с протокола *Modbus*. Этот коммуникационный протокол базируется на архитектуре клиент-сервер и был разработан фирмой *Modicon* для использования в контроллерах с программируемой логикой. Является фактическим стандартом для промышленных сетей передачи данных и широко применяется для организации связи промышленного электронного оборудования.

Для передачи данных в таком протоколе используются последовательные линии связи на базе интерфейсов RS-485, RS-422, RS-232, а также протоколы сетевого обмена *TCP/IP* (Transmission Control Protocol/Internet Protocol). Аппаратно протокол Modbus реализован в виде последовательного интерфейса, совместимого с протоколом RS-232, который интегрирован в стандартные

Modbus-порты в контроллерах Modicon. Сами контроллеры можно соединять между собой напрямую или посредством модемов. Обмен данными между контроллерами происходит по схеме "ведущий-ведомый". Только одно устройство (ведущий) может инициировать обмен данными, в то время как остальные устройства (ведомые) передают ведущему данные или выполняют указанные в запросе ведущего действия.

Как правило, ведущий реализован аппаратно как хост-процессор с набором модулей программирования, а ведомый представляет собой программируемый контроллер. Ведущий может обращаться к отдельному устройству или инициировать широковещательный запрос по шине Modbus. Ведомый, обнаружив в заголовке запроса свой адрес, отвечает ведущему, после чего начинается обмен данными. При широковещательном запросе ответы ведущему не возвращаются. Для передачи данных по протоколу Modbus используется единый простой формат передачи данных PDU, который является частью более широкой спецификации ADU.

Шина Modbus может работать в одном из трех режимов:

- ❒ режим RTU, который используется для передачи данных по последовательным линиям связи с интерфейсами RS-485, RS-422 и RS-232;
- ❒ режим ASCII, в котором также используются интерфейсы RS-485, RS-422 и RS-232;
- ❒ режим сетевой передачи, в котором используется протокол TCP.

На шине Modbus, в которой используются сигнальные линии интерфейсов RS-232 и RS-485, можно адресовать 247 ведомых устройств.

Еще одним протоколом обмена данными, о котором хотелось бы упомянуть, является протокол Profibus, в котором широко используются последовательные интерфейсы RS-485 (версия FMS). Profibus объединяет технологические и функциональные особенности последовательной связи полевого уровня. Он позволяет объединять разрозненные устройства автоматизации в единую систему на уровне датчиков и приводов.

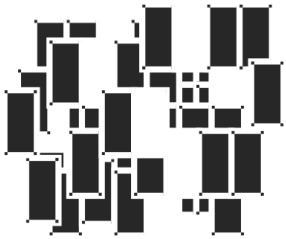
Profibus использует обмен данными между ведущим и ведомыми устройствами (протоколы DP и PA) или между несколькими ведущими устройствами (протоколы FDL и FMS). Требования пользователей к получению открытой, независимой от производителя, системы связи базируется на использовании стандартных протоколов Profibus.

Протокол *Profibus-FMS* разработан для связи ведущих устройств (контроллеров и ПК) друг с другом. Этот протокол используется там, где степень функциональности более важна, чем быстрое время реакции системы, и применяется на высоком уровне. Линия передачи информационных сигналов соответствует стандарту RS-485.

Протокол *Profibus-DP* спроектирован для высокоскоростной передачи данных между ведущим (контроллер) и оконечными устройствами (датчики, первичные преобразователи) сети и применяется на нижнем уровне системы промышленной автоматизации. Передача данных, как и для Profibus-FMS, основана на протоколе RS-485. Скорость передачи прямо зависит от протяженности сегмента сети и может изменяться от 100 Кбит/с для расстояния 1200 м до 12 Мбит/с для расстояния 100 м.

Существуют и другие популярные протоколы обмена данными с использованием последовательных интерфейсов, но далее мы будем рассматривать только те, о которых только что упомянули и которые базируются на интерфейсах RS-232 и RS-485.

Глава 2



Стандарты и протоколы последовательного интерфейса

Обмен данными по последовательным каналам связи регламентируется несколькими стандартами (протоколами, интерфейсами), определяющими аппаратно-программную архитектуру систем последовательной передачи данных. Основным протоколом последовательного обмена данными является RS-232. В соответствии с этим протоколом функционирует множество коммуникационных устройств, включая последовательные, или, по-другому, сериальные порты (serial ports) компьютерных систем. В подавляющем большинстве случаев для физической связи устройств, работающих по этому стандарту, используются 9-контактные разъемы, хотя иногда можно встретить и 25-контактные. Передача-прием данных по RS-232 осуществляется последовательно, бит за битом, откуда, собственно, и произошло название "последовательный порт".

Последовательный порт — один из наиболее ранних интерфейсов, разработанных для использования обмена данными. Последовательные порты компьютерных систем обычно известны под названием COM-портов. Обмен данными через последовательные порты осуществляется с использованием специальных чипов, известных под названием UART (Universal Asynchronous Receiver/Transmitter). Скорость обмена данными по последовательному порту может достигать 115 Кбит/с (или иначе в английском варианте kbps), хотя последние разработки позволяют увеличить эту скорость до 460 Кбит/с.

В настоящее время последовательные порты используются, в основном, для подключения модемов и специальных терминальных устройств. Довольно часто они применяются для настройки и отладки различного оборудования. Более новые скоростные технологии, такие, например, как USB и Fireware, с

каждым днем все больше ограничивают применение последовательных интерфейсов. Тем не менее, последовательные порты все еще широко используются как в уже выпущенном оборудовании, так и в разрабатываемом.

Посмотрим, каким образом осуществляется обмен данными через последовательные порты. Принципиально возможны два способа обмена данными: синхронный и асинхронный.

При *синхронном* режиме передачи данных и передатчик, и приемник должны работать на одинаковой частоте. В начале обмена передатчик посылает приемнику последовательность синхробит, за которой следуют посылки бит. При синхронном обмене прием-передача данных может осуществляться на значительно более высоких скоростях, чем при асинхронном обмене, поскольку не нужно передавать старт-бит, стоп-бит, бит паритета при передаче одного байта. Недостатком синхронного обмена является то, что и передатчик, и приемник должны обладать высокой стабильностью частоты, поскольку даже небольшое рассогласование в частотах приведет к появлению быстро накапливающейся ошибки.

При *асинхронной* передаче каждому байту данных предшествует *старт-бит*, за ним следуют биты данных, после них может передаваться *бит паритета* (четности), и в завершение посылки данных передается *стоп-бит*, гарантирующий определенную выдержку времени между соседними посылками. Структуру посылки можно представить так, как показано на рис. 2.1.

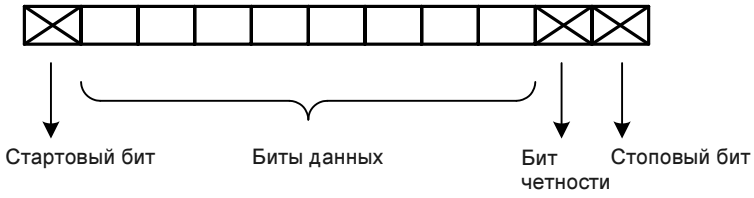


Рис. 2.1. Структура посылки данных

Формат данных последовательного порта обычно представляют в форме записи:

```
количество_бит_данных-тип_четности-количество_стоповых_бит
```

Например, запись 8-N-1 интерпретируется как посылка данных, содержащая 8 бит данных, без контроля четности и с одним стоповым битом.

Запись 7-E-2 интерпретируется как посылка, содержащая 7 бит данных, с контролем четности и с двумя стоповыми битами.

Биты данных чаще всего интерпретируются как биты ASCII-символа. Стартовый бит следующей посылки может передаваться в любой момент времени после окончания стопового бита предыдущей посылки. Количество бит данных может быть равным 5, 6, 7 или 8 битам, а количество стоповых битов может быть 1, 1.5, 2. Бит четности может отсутствовать.

Асинхронный обмен данными может осуществляться на одной из скоростей: 50, 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600 и 115200 бит/с. Для обмена данными в асинхронном режиме необходимо установить одинаковые параметры приемника и передатчика по скорости, количеству бит данных, проверке четности и количеству стоповых бит.

В компьютерных системах наиболее широко используется стандарт асинхронного обмена данными RS-232. Он применяется как для обмена данными через последовательные (COM) порты компьютера, так и для связи автономных периферийных устройств между собой. Рассмотрим интерфейс более подробно.

2.1. Интерфейс RS-232

На аппаратном уровне интерфейса RS-232 используются несимметричные приемопередающие устройства, при этом уровень сигнала отсчитывается относительно общего провода (сигнальной земли). Аппаратная часть интерфейса RS-232 не обеспечивает гальванической развязки устройств по цепям питания (в отличие от другого интерфейса, известного под названием *"токовая петля"*).

Логические уровни сигналов интерфейса находятся в диапазонах от –3 до –25 В (логическая "1") и от +3 до +25 В (логический "0"). Диапазон от –3 до +3 В соответствует неактивному состоянию (зона нечувствительности или гистерезис). Обычно сигнал, соответствующий уровню логической "1", называют "пробелом" (Space), а сигнал, соответствующий уровню логического "0" — "маркером" (Mark). Дальность связи при использовании интерфейса RS-232 достигает 15 м, поэтому чаще всего этот протокол используют в лабораторных и учебных системах обмена данными, а также при наладке и тестировании коммуникационного оборудования.

В основе асинхронного обмена данными, который используется при функционировании интерфейса RS-232 (как, впрочем, и многих других стандартов), положен принцип обработки отдельных кадров (*фреймов*) данных, передаваемых между стартовым и стоповым(и) битами.

Интерфейс RS-232 использует несколько сигнальных линий, но для иллюстрации передачи-приема данных мы посмотрим то, что происходит на любой из линий TD или RD. Для этого проанализируем процесс передачи-приема байта данных, который в упрощенном виде можно представить так, как показано на рис. 2.2.

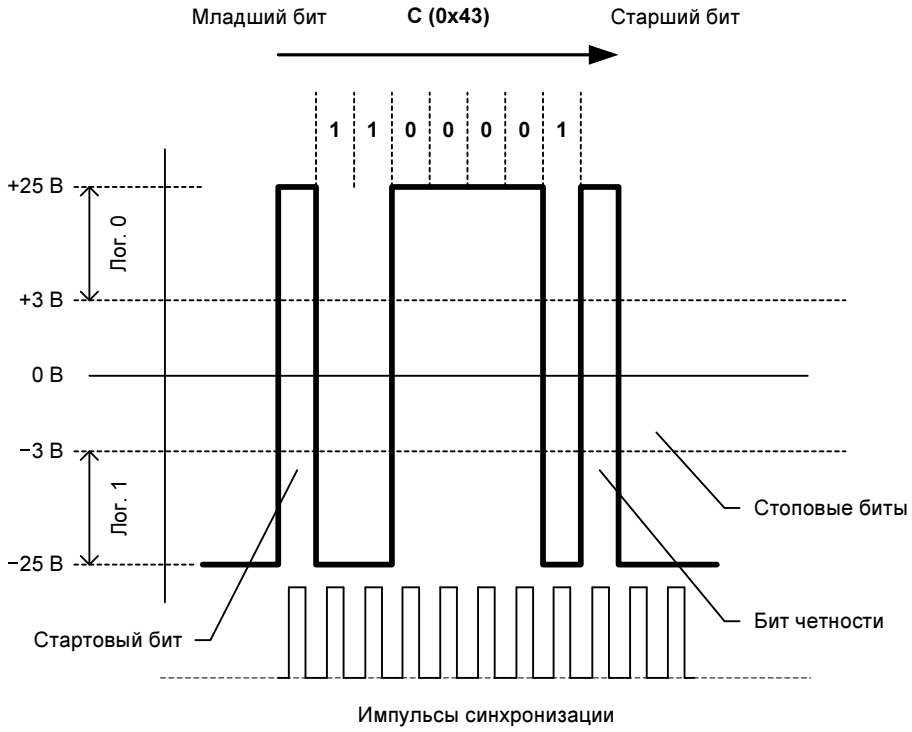


Рис. 2.2. Передача байта данных

На рисунке демонстрируется передача символа C в асинхронном режиме: 7 бит данных, бит четности и два стоповых бита. При этом каждому байту данных предшествует стартовый бит, за которым следуют биты данных (в данном случае 7 бит, представляющих символ C). Первым передается младший бит, последним — старший.

После передачи битов данных передается бит четности (в данном случае равный 0), и завершают передачу байта данных два стоповых бита, гарантирующих определенную выдержку времени между соседними посылками.

Обратите внимание на то, что логические уровни сигналов инверсные по отношению к полярности напряжения сигналов, т. е. отрицательным перепадам напряжения соответствует логическая "1", а положительным — логический "0".

Стартовый бит следующей посылки может передаваться в любой момент времени после окончания стопового бита предыдущей посылки, поэтому между двумя последовательными посылками данных могут быть паузы произвольной длительности.

Синхронизация приемника осуществляется с помощью стартового бита передатчика — при этом приемник и передатчик должны работать на одной и той же тактовой частоте (см. рис. 2.2). По спаду стартового бита передатчика генератор синхроимпульсов приемника сбрасывается и начинает формировать импульсы синхронизации, которые через программируемый делитель частоты и схемы синхронизации управляют приемом данных.

Аппаратно прием и передача данных через интерфейс RS-232 обычно реализуются в многофункциональном контроллере, который сокращенно называют UART. Это общее название для данного типа чипов, хотя выпускается очень много разновидностей этого устройства. Наиболее распространенными контроллерами приема-передачи по последовательному порту являются устройства, совместимые с серией 8250, которая включает следующие чипы: 16450, 16550, 16650, 16750.

Для безошибочного приема каждый бит данных должен захватываться в середине интервала времени, когда он является действительным (см. рис. 2.2). С возрастанием скорости передачи синхронизация передатчика и приемника усложняется — начинают сказываться паразитные емкости электронных узлов, затягивающие фронты синхроимпульсов, а также распределенная индуктивность физической линии передачи.

На практике интерфейс RS-232 реализован в виде сигнальных линий, имеющих следующее назначение:

- SG (Signal Ground) — сигнальная земля (относительно нее отсчитываются уровни сигналов);
- TD (Transmit Data) — выход передатчика (TXD);
- RD (Receive Data) — вход приемника (RXD);
- RTS (Request-To-Send) — запрос обмена данными, который выставляется контроллером последовательного порта (UART);

- CTS (Clear-To-Send) — указывает на готовность модема или терминального устройства к обмену данными;
- DTR (Data Terminal Ready) — указывает модему или другому устройству, что контроллер последовательного порта (UART) готов установить соединение;
- DSR (Data Set Ready) — указывает контроллеру (UART) последовательного порта на то, что периферийное устройство готово установить соединение;
- DCD (Data Carrier Detect) — входной сигнал от удаленного модема или устройства. Когда модем или другое устройство обнаруживает появление несущей на другом конце линии, то линия становится активной;
- RI (Ring Indicator) — вход индикатора вызова.

Сигналы RD и TD, определяющие прием-передачу данных, мы рассмотрели на примере передачи-приема байта данных. Обратите внимание на то, что линии данных являются последовательными, в то время как сигнальные линии RTS, CTS, DCD, DSR, DTR, RTS и RI являются, по сути, параллельными.

Для соединения устройств, работающих с использованием данного интерфейса, применяется, как уже упоминалось, 9-контактный разъем, выводы которого соответствуют сигналам, указанным в табл. 2.1.

Таблица 2.1. *Сигналы и выводы интерфейса RS-232 для 9-контактного разъема*

Номер вывода на разъеме	Сигнал	Номер вывода на разъеме	Сигнал
3	TD	5	SG
2	RD	1	DCD
7	RTS	4	DTR
8	CTS	9	RI
6	DSR		

Эти же сигналы присутствуют и на COM-порту компьютера, к которому подсоединено какое-либо устройство, например, модем.

Для соединения двух устройств по последовательному каналу используются так называемые *"нуль-модемные" соединения*. В зависимости от аппаратно-программной конфигурации системы обмена данными может использоваться как полнофункциональное соединение (рис. 2.3, *а*), так и максимально упрощенный вариант (рис. 2.3, *б*), который довольно часто используется как для диагностических целей, так и для управления системами промышленной автоматики на базе специализированных модулей, например, микроконтроллеров.

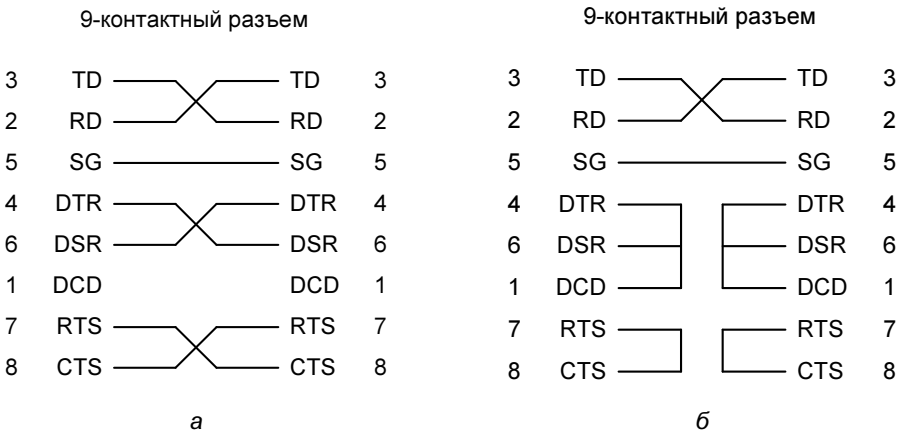


Рис. 2.3. Нуль-модемное соединение:
а — полнофункциональное соединение, *б* — упрощенный вариант

Еще один, весьма распространенный случай, когда разработчик должен проверить функционирование приложений, работающих с последовательными портами. Для быстрой проверки таких программ можно воспользоваться так называемым *интерфейсом обратной связи* (loopback interface), схема которого приведена на рис. 2.4.

Сигналы интерфейса RS-232 позволяют управлять потоком данных в зависимости от состояния передатчика и приемника. При этом могут использоваться несколько протоколов обмена данными. Хочу сразу же уточнить, что эти протоколы не касаются базовых принципов передачи данных, рассмотренных ранее, их назначение иное — они позволяют синхронизировать устройства, имеющие различные скорости приема-передачи данных, различные размеры буферов памяти для хранения данных. Например, принтер, получающий данные через последовательный порт, может откладывать их

прием до окончания обработки предыдущей порции данных, сигнализируя об этом передатчику.

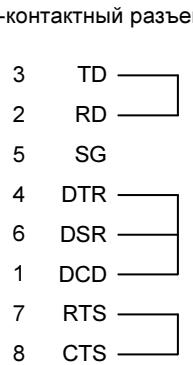


Рис. 2.4. Схема интерфейса обратной связи

Для управления потоком данных используются аппаратный или программный протоколы. Рассмотрим некоторые, наиболее часто используемые протоколы управления обменом данными:

- аппаратный протокол RTS/CTS — используется для соединения принтера с компьютерной системой или для соединения компьютеров, при этом нуль-модемный кабель использовать нельзя;
- аппаратный протокол DTR/DSR — аналогичен RTS/CTS, но использует другие сигналы;
- программный протокол XON/XOFF — используется в двунаправленных каналах обмена при буферизованном обмене данными. При полном заполнении буфера приемника передача данных прекращается и ее возобновление возможно после обработки данных в буфере приемника. Приемник останавливает передачу данных передатчиком, посылая ему команду `XOFF`, и возобновляет ее посылкой команды `XON`;
- программный протокол ACK — синхронизация получения одного байта данных или группы байтов. Приемник посылает передатчику команду `ACK`, в ответ на которую передатчик посылает приемнику байт (или пакет байтов).

Рассмотрим примеры аппаратно-программной реализации протокола RS-232 в компьютерных системах обмена данными.

2.2. Примеры аппаратно-программной реализации протокола RS-232

К настоящему времени разработано и используется множество различных аппаратно-программных конфигураций интерфейса RS-232. Рассмотрим наиболее распространенные из них. Первая задача, которую приходится решать разработчикам интерфейса — согласование уровней напряжения логических сигналов. Уровень напряжения логической "1" по спецификации протокола RS-232 должен находиться в пределах от –3 до –25 В, а уровень логического "0" — в диапазоне от +3 до +25 В, при этом уровни напряжений в диапазоне от –3 до +3 В являются неопределенными. Внутренние цифровые схемы большинства устройств работают с уровнями напряжений от 1,8 до 5 В, поэтому для передачи сигналов интерфейса RS-232 в линию необходимо использовать либо отдельный, либо интегрированный в устройство преобразователь уровня сигналов. Например, микросхема асинхронного приемопередатчика (UART) персонального компьютера работает с уровнями сигналов, соответствующих TTL-логике, что требует использования буферного приемопередатчика.

Наибольшее распространение в аппаратных интерфейсах RS-232 получил буферный приемопередатчик сигналов интерфейса MAX232. Эта микросхема была разработана относительно давно, но в настоящее время все еще очень широко используется при разработке интерфейса RS-232 и считается промышленным стандартом для подобного типа устройств. Микросхема MAX232 позволяет связать различные устройства по последовательному каналу (рис. 2.5).

Здесь продемонстрирована организация канала связи между последовательным портом персонального компьютера и внешним устройством на базе микроконтроллера/микропроцессора при помощи MAX232.

Кристалл MAX232 включает внутренний удвоитель-инвертор напряжения на переключаемых конденсаторах с подкачкой заряда (они подключаются к выводам C1+, C1–, C2+ и C2–). Это позволяет получить +10 В и –10 В из одного источника питания +5 В. Сигналы с уровнями ±10 В используются для обмена данными по интерфейсу RS-232. В одном корпусе микросхема содержит два буферных приемника и два передатчика сигнала. В настоящее время используются и более совершенные версии этого кристалла, причем некоторые из микросхем, например, DS275, не содержат навесных компонентов. Что же касается емкости конденсаторов, то для некоторых модификаций MAX232

можно использовать и конденсаторы меньшей емкости, например, 1 мкФ. В наших последующих примерах мы будем использовать стандартную микросхему MAX232.

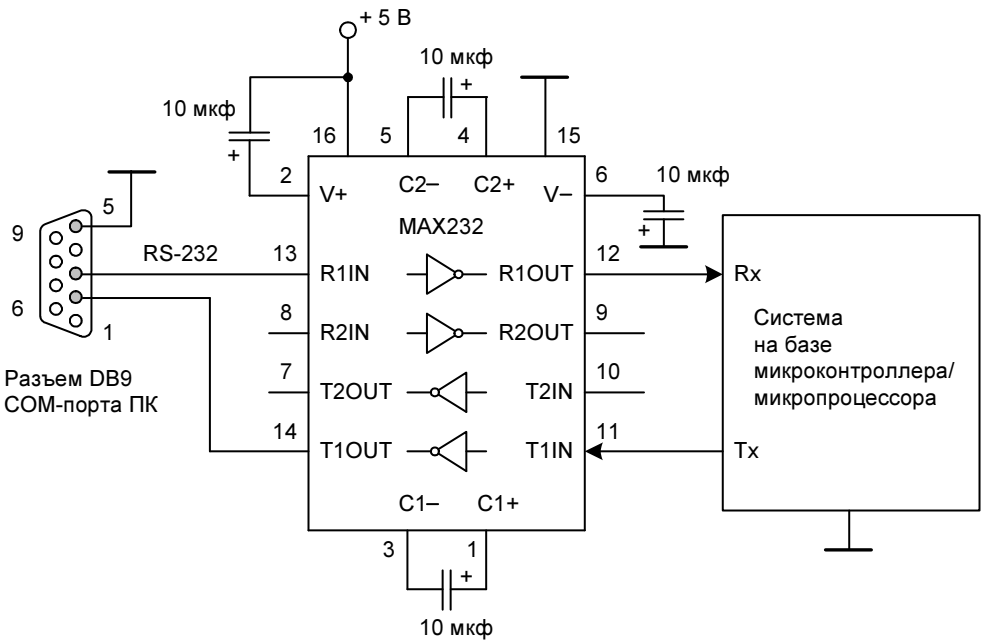


Рис. 2.5. Реализация интерфейса на базе RS-232 между двумя устройствами

Для передачи данных по последовательному интерфейсу в кристаллы большинства промышленных микроконтроллеров или микропроцессоров интегрируется модуль последовательного интерфейса. Сигналы этого интерфейса имеют уровни TTL-логики, а количество сигнальных линий и линий аппаратного контроля может варьироваться в зависимости от круга возможных задач, для которых предназначено устройство. Например, в большинстве микроконтроллеров серии 8051/8052 имеется, как правило, один стандартный интерфейс последовательного обмена, в котором используются только две сигнальные линии — TxD и RxD. В этом случае обеспечивается программный контроль потока данных, чего вполне достаточно для простых систем обмена данными. В микроконтроллерах PIC24 фирмы Microchip на кристалле может находиться несколько модулей последовательной передачи данных, работающих независимо друг от друга, причем для обмена данными могут

использоваться и линии аппаратного управления потоком данных. Кроме того, наличие нескольких независимых модулей последовательного обмена данными позволяет организовать работу в "чистом" полнодуплексном режиме с отдельным управлением потоком данных по двум каналам.

При небольших расстояниях (до 1,5—2 м) и небольших скоростях два подобных устройства могут обмениваться данными без использования буферных формирователей, однако при увеличении расстояния и скорости обмена без драйверов интерфейса такие системы будут работать с ошибками.

Далее мы рассмотрим различные демонстрационные проекты обмена данными по интерфейсу RS-232, в которых будут проанализированы схемы обмена данными на уровне сигналов протокола RS-232. В подавляющем большинстве современных приложений интерфейс RS-232 используется, как правило, в лабораторных системах управления и контроля. Такие системы состоят из двух частей: компьютерной системы (обычно это персональный компьютер) и внешнего устройства управления и контроля на базе специализированной системы с микроконтроллером или микропроцессором. В качестве процессорного модуля таких внешних систем очень часто используются микроконтроллеры серий 8051/8052 или PIC-микроконтроллеры.

В большинстве демонстрационных проектов этой главы будут применяться микроконтроллеры 8051/8052. Такой выбор обусловлен несколькими причинами. Во-первых, во всех проектах будут задействованы те или иные сигнальные линии интерфейса RS-232, функционирование которых проще всего показать с помощью аппаратно-программной системы на микроконтроллере. Во-вторых, устройства 8052 очень широко распространены в промышленных и учебных системах управления и контроля, имеют интуитивно понятную систему команд и очень легко программируются для последовательного обмена данными на языке C.

И, наконец, интерфейс последовательного обмена микроконтроллеров включает две базовые линии, TxD и RxD, для передачи и приема данных, к которым легко подключаются популярные драйверы линии, такие, как MAX232. В отдельных проектах мы будем использовать также и мощные 16-битовые микроконтроллеры PIC24.

Пример простейшей системы обмена данными по интерфейсу RS-232 между двумя микроконтроллерами 8052 показан на рис. 2.6.

Здесь используется нуль-модемная конфигурация, в которой задействованы две сигнальные линии передачи и приема данных. При такой конфигурации выход TxD одного устройства подключается ко входу RxD другого устройства, и наоборот, вход RxD одного устройства подключается к выходу TxD

другого устройства. Кроме того, общие провода обоих устройств должны быть соединены вместе. Для согласования TTL-уровней сигналов микроконтроллеров DD1—DD2 с уровнями интерфейса RS-232 используются два стандартных драйвера линии MAX232. В этой схеме отсутствует аппаратный и программный контроль обмена данными, поэтому подобную конфигурацию можно использовать лишь в диагностических целях или при работе устройств, в которых не возникнет "зависание" оборудования при приеме или передаче данных.

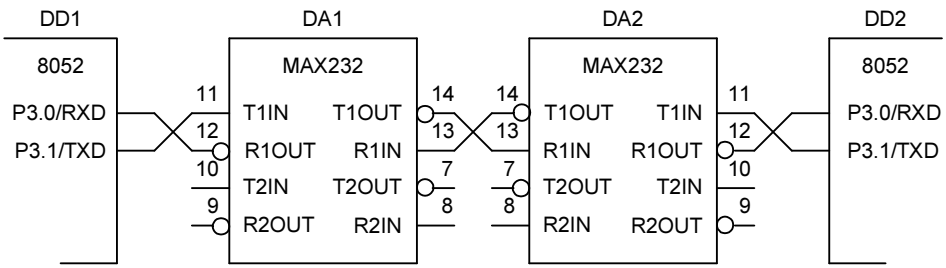


Рис. 2.6. Система обмена данными по интерфейсу RS-232 между микроконтроллерами

Программное обеспечение для обмена данными по такой схеме определяется конкретными особенностями работы системы. Предположим, что микроконтроллер DD1 является передатчиком данных, а микроконтроллер DD2 — их приемником. Программа для передатчика на языке Keil C51 будет включать следующий код, представленный в листинге 2.1.

Листинг 2.1. Программа передачи строки данных микроконтроллеру по интерфейсу RS-232

```
#include <REG52.H>
#include <stdio.h>

void main(void)
{
 char *s1 = "Test String for RS-232 Receiver!";

 SCON = 0x50;
 TMOD |= 0x20;
```

```
 TH1 = 0x0FD;
 TR1 = 1;
 TI = 1;

 printf(s1);
 while(1);
 }
```

Программа, записанная в устройство DD1, посылает строку данных микроконтроллеру DD2. В передатчике последовательного порта задействована всего одна сигнальная линия TXD, по которой данные передаются на вход передачи T1IN драйвера MAX232 интерфейса RS-232 (DA1 на схеме рис. 2.6) и далее через буфер приемника DA2 на вход RXD микроконтроллера 8052 (DD2). Скорость передачи и приема данных устанавливается равной 9600 бод.

Операторы

```
SCON = 0x50;
TMOD |= 0x20;
TH1 = 0x0FD;
TR1 = 1;
TI = 1;
```

позволяют настроить скорость передачи данных 9600 бод при стандартной тактовой частоте микроконтроллера 11,059 МГц.

Далее приводится исходный текст программы приемника (микроконтроллер DD2).

Листинг 2.2. Программа приема строки данных микроконтроллером по интерфейсу RS-232

```
#include <REG52.H>
#include <stdio.h>

void main(void)
{
 char buf[64];
 char *pbuf = buf;
```

```
 SCON = 0x50;
 TMOD |= 0x20;
 TH1 = 0x0FD;
 TR1 = 1;
 TI = 1;

 while (1)
  {
   *pbuf = getchar();
   if (*pbuf == 0)
     break;
   pbuf++;
  }
  printf("%s", buf);
}
```

Программа побайтово принимает строку данных из последовательного порта в цикле `while (1)` с помощью функции `getchar`. Принятые байты записываются в буфер `buf`, а прием байтов выполняется до обнаружения в потоке данных нулевого символа. По окончании приема принятая строка отправляется обратно устройству-передатчику. Группа операторов

```
SCON = 0x50;
TMOD |= 0x20;
TH1 = 0x0FD;
TR1 = 1;
TI = 1;
```

выполняет ту же функцию, что и в передатчике — она используется для настройки скорости приема 9600 бод для стандартной тактовой частоты микроконтроллера 11,059 МГц. Скорость приема должна совпадать со скоростью передачи в передатчике данных.

В роли передатчика данных может выступать последовательный порт персонального компьютера. В этом случае получим простую систему управления на базе протокола RS-232. В такой системе микроконтроллер будет получать данные от программы, работающей под управлением какой-либо операционной системы, и выполнять соответствующий алгоритм их обработки. Очень часто данные с последовательного порта персонального компьютера переда-

ются во внешнее устройство по интерфейсу RS-232, где используются для управления исполнительными устройствами. Пример такой системы управления на базе популярного микроконтроллера 8051 показан на рис. 2.7.

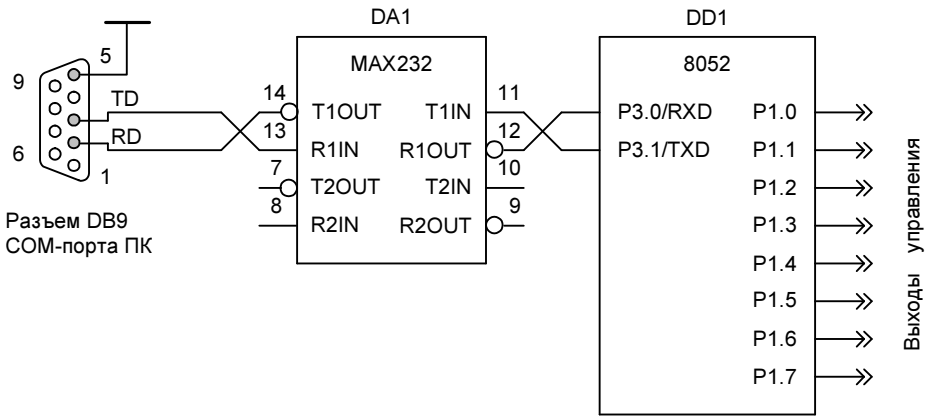


Рис. 2.7. Система управления на базе интерфейса RS-232

В этой системе данные, полученные по последовательному порту, выводятся в порт P1 микроконтроллера и могут управлять различными нагрузками. Исходный текст программы управления, записанной в микроконтроллере 8052, показан в листинге 2.3.

Листинг 2.3. Программа управления внешними устройствами по интерфейсу RS-232

```
#include <REG52.H>
#include <stdio.h>

void SerialISR(void) interrupt 4
{
 if (RI)
  {
   RI = 0;
   P1 = SBUF;
  }
}
```

```
void main(void)
{
 P1 = 0x0;
 SCON = 0x50;
 TMOD |= 0x20;
 TH1 = 0x0FD;
 TR1 = 1;

 EA = 1;
 ES = 1;
 while(1);
 }
```

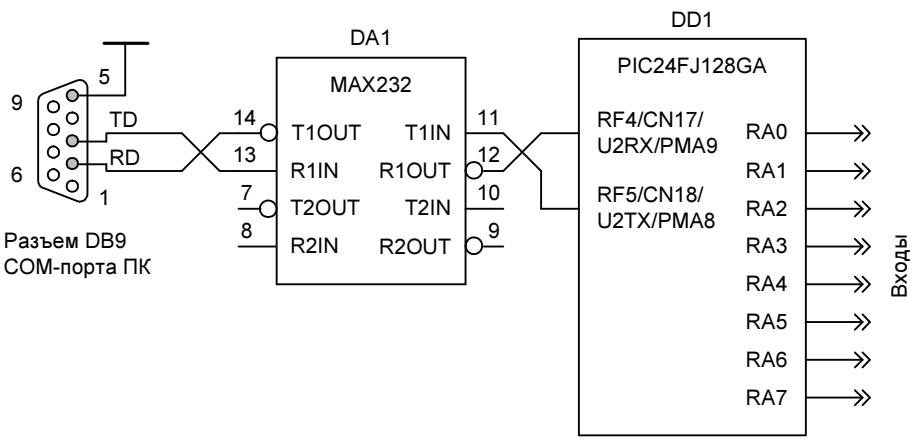


Рис. 2.8. Передача байта данных в ПК по интерфейсу RS-232

Для приема байта данных в этой программе используется прерывание последовательного порта с функцией-обработчиком `SerialISR`. Поскольку прерывание последовательного порта может быть вызвано установкой одного из флагов TI (передачи) или RI (приема), то здесь анализируется только флаг приема RI. Байт данных, принятый по последовательному порту, помещается в регистр буфера приемопередатчика SBUF. Содержимое этого регистра затем записывается в порт ввода-вывода P1, настроенный на вывод данных. Сигналы порта могут использоваться для управления внешними исполни-

тельными устройствами. Первые пять операторов основной программы инициализируют последовательный порт микроконтроллера, а операторы

```
EA = 1;
ES = 1;
```

разрешают прерывание последовательного порта.

Расширить возможности управления можно с помощью 16-битовых микроконтроллеров PIC24, которые имеют несколько модулей последовательного ввода-вывода и большое количество цифровых портов ввода-вывода. Схема чтения 8 цифровых датчиков через входы порта A микроконтроллера PIC24 и передача байта данных по интерфейсу RS-232 в персональный компьютер показаны на рис. 2.8.

Исходный текст программы микроконтроллера PIC24, работающей с этим интерфейсом, представлен в листинге 2.4.

Листинг 2.4. Передача входных сигналов в параллельном коде по интерфейсу RS-232

```
#include <p24fj128ga010.h>
#include <stdio.h>

_CONFIG2( FCKSM_CSDCMD & OSCIOFNC_ON & POSCMOD_HS & FNOSC_PRI )

#define SYSCLK        8000000
#define BAUDRATE2     9600
#define BAUDRATEREG   SYSCLK /8 / BAUDRATE2-1

#define t1            3
#define PREG          SYSCLK / 2 / 256 * t1

#define UART2_TX_TRIS   TRISFbits.TRISF5
#define UART2_RX_TRIS   TRISFbits.TRISF4

void UART2Init()
{
  UART2_TX_TRIS = 0;
  UART2_RX_TRIS = 1;
```

```
  U2BRG = BAUDRATEREG;
  U2MODE = 0;
  U2MODEbits.BRGH = 1;
  U2MODEbits.UARTEN = 1;
  U2STA = 0;
  U2STAbits.UTXEN = 1;
  IFS1bits.U2RXIF = 0;
}

void  UART2PutChar(char Ch)
{
  while(U2STAbits.UTXBF == 1);
  U2TXREG = Ch;
}

char tmp;
char buf[32];
int i1, bytes;

void __attribute__ ((__interrupt__)) _T1Interrupt(void)
{
  _T1IF = 0;
  tmp = PORTA;
  bytes = sprintf(buf, "0x%X ", tmp);
  for (i1 = 0; i1 < bytes; i1++)
  UART2PutChar(buf[i1]);
}

void main()
{
  TRISA = 0xffff;
  UART2Init();
  _T1IF = 0;
  _T1IE = 1;
```

```
  PR1 = PREG;
  TMR1 = 0;
  T1CON = 0x8030;

  while (1)
   {
   }
}
```

Программа каждые 3 сек считывает один байт данных с 8-ми дискретных входов порта A микроконтроллера, затем преобразует данные в шестнадцатеричный формат и посылает их посредством интерфейса RS-232 в персональный компьютер. Эти операции выполняются функцией-обработчиком прерывания `_T1Interrupt` Таймера 1, который используется для отсчета 3-секундных временных интервалов. Скорость передачи данных выбрана здесь равной 9600 бод при тактовой частоте микроконтроллера 8 МГц.

Эти примеры построения интерфейса RS-232 являются типичными и используются в большинстве систем обработки данных. В качестве внешнего устройства здесь выступает система на базе микроконтроллера/микропроцессора, которая, в свою очередь, может управлять другими внешними устройствами. Микроконтроллеры обеспечивают первичную обработку данных и передают их по последовательному каналу системе более высокого уровня, например, ПК, либо принимают данные от другой системы.

Недостатком протокола RS-232 является то, что с его помощью обмен данными может выполняться только между двумя устройствами. Протокол не определяет, каким образом можно идентифицировать устройство, отправляющее или принимающее данные. Интерфейс RS-232 не определяет дополнительных линий выбора адреса устройства, тем не менее, обмен данными между несколькими устройствами можно организовать программными средствами. Например, при передаче данных сначала можно посылать байт адреса, а затем — один или несколько байтов данных. Программное обеспечение всех устройств должно анализировать первый байт передаваемой последовательности и сравнивать его с определенным значением, которое считается адресом устройства. Для подключения нескольких устройств можно использовать многоканальные драйверы линии или воспользоваться несколькими микросхемами MAX232. На рис. 2.9 представлен пример конфигурации из трех устройств, соединенных по интерфейсу RS-232.

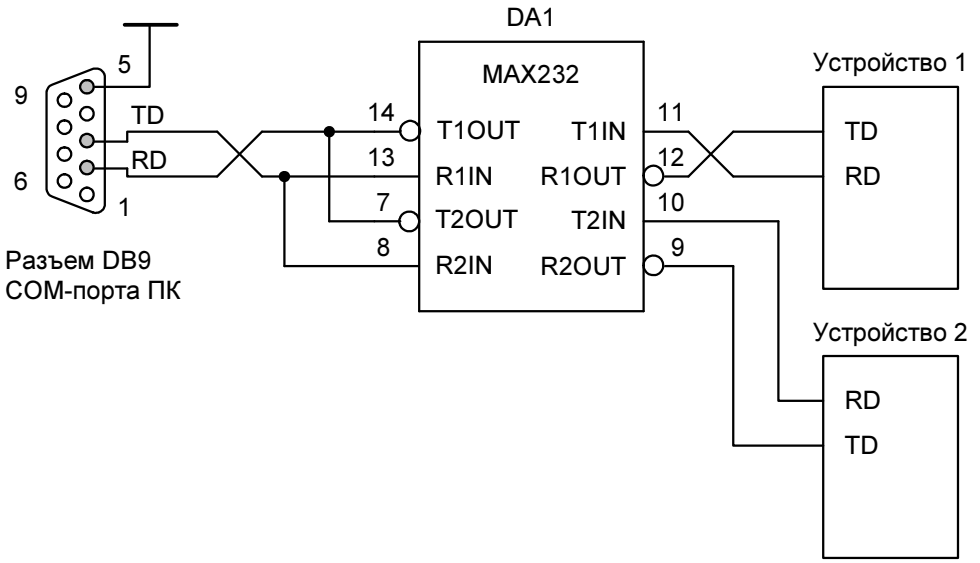


Рис. 2.9. Подключение двух устройств к интерфейсу RS-232 персонального компьютера

Здесь данные от последовательного порта персонального компьютера передаются на два внешних устройства посредством микросхемы MAX232, в которой задействованы оба канала передачи. Подобным образом можно организовать соединение и большего числа устройств по RS-232. Такие схемы могут пригодиться при создании небольших лабораторных и учебных сетей. Следует лишь учитывать, что длина соединений не должна превышать нескольких метров, поскольку при дальнейшем увеличении расстояния вследствие затухания сигнала и помех надежность работы такой сети будет уменьшаться.

2.3. Управление потоком данных

До сих пор мы рассматривали простейшие системы обмена данными, в которых были задействованы линии передачи (TxD) и приема (RxD) данных. При этом предполагалось, что как приемник, так и передатчик работают без сбоев и задержек. Это справедливо для несложных систем диагностики и тестирования на базе протокола RS-232. В большинстве реальных промышленных и лабораторных систем ситуации потери данных вследствие "зависания" или

перегрузки процессора устройства не так уж и редки. Для обеспечения большей надежности при обмене данными разработано несколько методов контроля и управления потоком данных. Выделяют системы с *программным* (software control) и *аппаратным управлением* потоком данных (hardware control). Программный метод управления потоком данных реализован в протоколе XON/XOFF, а аппаратный метод управления — в протоколе RTS/CTS (DTR/DSR). Рассмотрим методы практической реализации обоих методов управления потоком данных.

2.3.1. Программный метод управления потоком данных

Реализация программного метода управления (метод XON/XOFF) не требует наличия дополнительных сигнальных линий, поэтому обмен данными можно выполнять и в простейшей нуль-модемной конфигурации по двум линиям TxD и RxD. Для использования метода XON/XOFF протоколом предусмотрено два специальных байта, которые так и называются — байт XON и байт XOFF. Байт XON имеет десятичный код 17, а XOFF — 19. Байт XON используется для разрешения передачи данных, а XOFF — для остановки. На клавиатуре эти коды эмулируются комбинациями клавиш <Ctrl>+<Q> (XON) и <Ctrl>+<S> (XOFF). Применение метода XON/XOFF для управления потоком данных не вызывает затруднений. Для остановки передачи данных приемник посылает байт XOFF передатчику, а для возобновления передачи — байт XON (рис. 2.10).

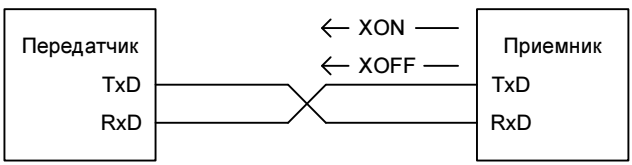


Рис. 2.10. Программное управление потоком данных

У программного метода управления потоком данных есть два недостатка. Один из них заключается в том, что использование дополнительных управляющих байтов снижает общую скорость передачи данных. Второй недостаток более существенный и связан с неправильным функционированием программного обеспечения, управляющего потоком данных. Если, например,

программное обеспечение некорректно отработает ситуацию переполнения буфера приема, то управляющие байты XON/XOFF могут быть потеряны, что приведет к существенным ошибкам при обмене данными.

Кроме того, существует вероятность потери управляющих байтов на линиях с высоким уровнем помех, что характерно для высокоскоростных линий передачи данных. По этой причине программный метод управления передачей данных используется на относительно низкоскоростных линиях обмена данными. Тем не менее, метод XON/XOFF, благодаря простоте реализации, довольно широко используется во многих системах передачи данных.

Рассмотрим практический метод реализации программного управления потоком данных в системе обмена данными по протоколу RS-232, аппаратная схема которой показана на рис. 2.11.

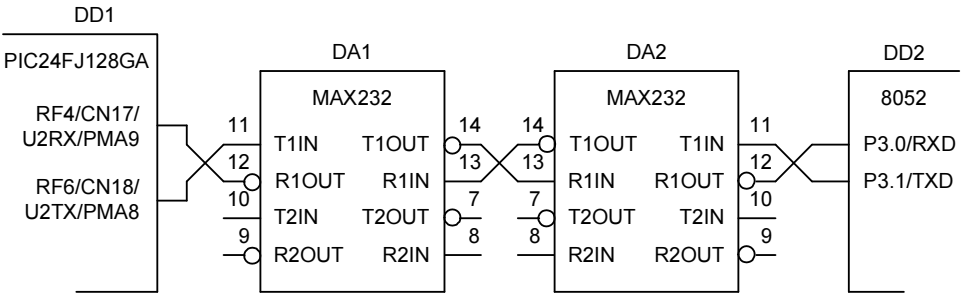


Рис. 2.11. Схема для демонстрации
метода программного управления потоком XON/XOFF

В этой схеме обмена данными система на микроконтроллере PIC24 (DD1) передает строку данных приемнику DD2 (микроконтроллер 8052) по интерфейсу RS-232. Согласование уровней сигналов интерфейса выполняют два драйвера линии MAX232. Программное обеспечение передатчика передает побайтово строку данных и анализирует байты данных, приходящие от приемника. Если приемник передает байт XOFF, то передача данных приостанавливается, если приемник передает байт XON, передача возобновляется.

Программное обеспечение приемника на микроконтроллере 8052 отсчитывает количество принятых от передатчика байтов и после получения 5-го байта посылает передатчику байт XOFF. По истечении интервала в 7 сек приемник посылает передатчику байт XON, что приводит к возобновлению передачи данных. Исходный текст программы передатчика на языке MPLAB C30 показан в листинге 2.5.

Листинг 2.5. Программа передачи строки данных по интерфейсу RS-232

```
#include <p24fj128ga010.h>

_CONFIG2( FCKSM_CSDCMD & OSCIOFNC_ON & POSCMOD_HS & FNOSC_PRI )

#define SYSCLK       8000000
#define BAUDRATE2    9600
#define BAUDRATEREG  SYSCLK /8 / BAUDRATE2-1

#define UART2_TX_TRIS   TRISFbits.TRISF5
#define UART2_RX_TRIS   TRISFbits.TRISF4

#define t1 0.5
#define PREG  SYSCLK /2 /256 * t1

#define XON  17
#define XOFF 19

void UART2Init()
{
  UART2_TX_TRIS = 0;
  UART2_RX_TRIS = 1;
  U2BRG = BAUDRATEREG;

  U2MODE = 0;
  U2MODEbits.BRGH = 1;
  U2MODEbits.UARTEN = 1;
  U2STA = 0;
  U2STAbits.UTXEN = 1;
  IFS1bits.U2RXIF = 0;
}

void  UART2PutChar(char Ch)
{
  while(U2STAbits.UTXBF == 1);
```

```
  U2TXREG = Ch;
}

char *str1 = "Testing XON/XOFF Flow control";
char Temp;
char Stop = 0;

void __attribute__ ((__interrupt__)) _T1Interrupt(void)
{
  _T1IF = 0;
  if (Stop == 0)
   {
    if (*str1 != 0)
    UART2PutChar(*str1++);
   }
}

void __attribute__ ((__interrupt__)) _U2RXInterrupt(void)
{
  _U2RXIF = 0;
  Temp = U2RXREG;
  if (Temp == XOFF)
   Stop = 1;
  if (Temp == XON)
   Stop = 0;
}

void main()
{

  UART2Init();
  _U2RXIF = 0;
  _U2RXIE = 1;

  _T1IF = 0;
  _T1IE = 1;
```

```
  PR1 = PREG;
  TMR1 = 0;
  T1CON = 0x8030;

  while (1)
  {
  }
}
```

Программа передачи байтов работает с каналом 2 (UART2) модуля последовательного обмена данными устройства PIC24FJ128GA. Для передачи одного байта данных используется прерывание Таймера 1 микроконтроллера с функцией-обработчиком `_T1Interrupt`, которое вызывается каждые 0,5 сек. Передача данных завершается при обнаружении нулевого символа, что свидетельствует о достижении конца строки. Функция-обработчик перед передачей очередного байта строки `str1` проверяет состояние переменной `Stop`. Если она установлена в 1, то это означает, что от приемника данных получен сигнал XOFF, и передача должна быть прекращена.

Если переменная `Stop` сброшена в 0, то передача очередного байта приемнику разрешена. Переменная `Stop` устанавливается или сбрасывается в зависимости от байта данных, полученного от приемника с помощью функции-обработчика прерывания `_U2RXInterrupt`, которая вызывается при поступлении байта данных в буфер приемника канала 2.

Функция `UART2PutChar` выполняет передачу байта в канал 2 последовательного обмена. Инициализация последовательного канала 2 модуля UART и настройка прерываний Таймера 1 и приемника канала 2 осуществляется в основной программе.

Программное обеспечение приемника на микроконтроллере 8052 написано на языке Keil C51. Исходный текст С-файла программы показан в листинге 2.6.

Листинг 2.6. Программа приема данных по интерфейсу RS-232

```
#include <REG52.H>

#define XON 17
#define XOFF 19
```

```
int cnt = 0;
unsigned long del = 100;

void Timer0_ISR(void) interrupt 1
{
  del--;
  if (del == 0)
   {
    del = 100;
    SBUF = XON;
    ET0 = 0;
   }
}

void SerialISR(void) interrupt 4
{
 if (RI)
  {
   RI = 0;
   cnt++;
   if (cnt == 5)
    {
     SBUF = XOFF;
     while (TI == 0);
     ET0 = 1;
     cnt = 0;
   }
  }
}

void main(void)
{
  SCON = 0x50;
  TMOD = 0x21;
  TH1 = 0x0FD;
  TR1 = 1;
  TI = 0;
```

```
  TH0 = 0;
  TL0 = 0;
  TR0 = 1;

  EA = 1;
  ES = 1;
  while(1);
 }
```

В этой программе для обработки приходящего по интерфейсу RS-232 потока байтов используется прерывание последовательного порта с функцией-обработчиком `SerialISR`. Функция-обработчик вызывается всякий раз при получении байта данных из последовательного порта по линии RxD. Собственно, никакой обработки байтов данных эта функция не выполняет, а только подсчитывает количество принятых байтов. Если получено 5 байтов данных, то передатчику посылается байт XOFF, останавливающий передачу данных.

Кроме того, в функции-обработчике `SerialISR` запускается таймер, который по истечению 7 сек передает байт разрешения XON передатчику, после чего передача данных возобновляется. Таким образом, после каждых пяти принятых байтов передача останавливается на 7 сек, после чего передаются очередные 5 байтов и т. д. Несмотря на некоторую искусственность приведенного примера программного управления потоком данных, он довольно наглядно демонстрирует механизм такого метода контроля.

2.3.2. Аппаратное управление потоком данных

В отличие от программного метода XON/XOFF, аппаратный метод управления потоком данных требует наличия дополнительных сигнальных линий, но это компенсируется определенными преимуществами. Аппаратный контроль потоком данных часто называют RTS/CTS-управлением. Для реализации этого метода используются сигнальные линии RTS/CTS. В классическом варианте компьютер или другое устройство устанавливает сигнал на линии RTS, указывая на готовность данных. Другое устройство определяет возможность получения информации и, если это возможно, устанавливает сигнал на линии CTS, инициируя тем самым начало передачи. При использовании нуль-модемных конфигураций *квитирование* (подтверждение готовности) передачи и приема данных реализуется немного по-другому. Из возможных вариан-

тов реализации широкое распространение получил метод, при котором линия RTS одного устройства соединяется с линией CTS другого устройства. В этом случае установленный сигнал RTS одного устройства будет свидетельствовать о готовности приема данных. Подобная конфигурация показана на рис. 2.12.

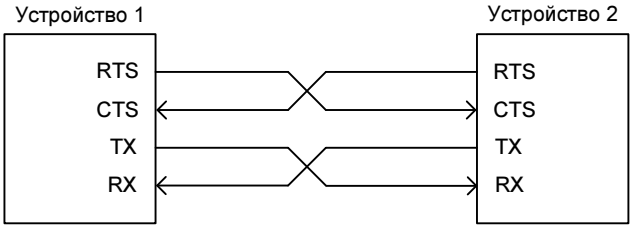


Рис. 2.12. Вариант аппаратного управления потоком данных

При такой упрощенной схеме аппаратного управления можно добиться высокой производительности обмена данными. Например, можно настроить программное обеспечение Устройства 2 таким образом, чтобы установка сигнала RTS Устройства 1 в активное состояние инициировала передачу данных от Устройства 2 к Устройству 1. Рассмотрим пример аппаратного управления потоком данных по интерфейсу RS-232. Аппаратная часть проекта показана на рис. 2.13.

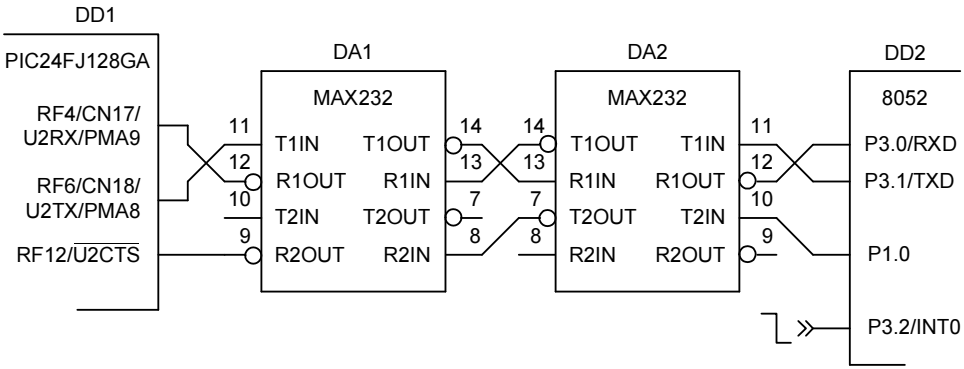


Рис. 2.13. Принципиальная схема аппаратного управления потоком данных

На этой схеме вывод RF12/nU2CTS (здесь и далее n в обозначениях сигналов указывает на инверсию сигнала) микроконтроллера PIC24 (DD1) выполняет функции стандартной сигнальной линии CTS передающего устройства, а вы-

вод P1.0 микроконтроллера 8052 (DD2) — функцию RTS-приемника данных (см. рис. 2.12). Принимающее устройство (DD2) синхронизирует начало передачи данных от устройства DD1 подачей низкого уровня напряжения сигнала на вход U2CTS при подаче сигнала прерывания на вход INT0 микросхемы приемника. Для формирования соответствующих уровней сигналов интерфейса RS-232 используются буферные приемопередатчики MAX232 (DA1—DA2).

Программа управления передачей данных микроконтроллером PIC24 (DD1) представлена в листинге 2.7.

Листинг 2.7. Программа передачи данных по интерфейсу RS-232 с синхронизацией

```
#include <p24fj128ga010.h>

_CONFIG2( FCKSM_CSDCMD & OSCIOFNC_ON & POSCMOD_HS & FNOSC_PRI )

#define SYSCLK        8000000
#define BAUDRATE2     9600
#define BAUDRATEREG   SYSCLK /8 / BAUDRATE2-1

#define UART2_TX_TRIS   TRISFbits.TRISF5
#define UART2_RX_TRIS   TRISFbits.TRISF4

void UART2Init()
{
  UART2_TX_TRIS = 0;
  UART2_RX_TRIS = 1;
  U2BRG = BAUDRATEREG;

  U2MODE = 0;
  U2MODEbits.UEN1 = 1;
  U2MODEbits.BRGH = 1;

  U2MODEbits.UARTEN = 1;
  U2STA = 0;
  U2STAbits.UTXISEL0 = 1;
```

```
  U2STAbits.UTXEN = 1;
  IFS1bits.U2RXIF = 0;
}

char *str = "RTS/CTS Flow Control test";

void __attribute__ ((__interrupt__)) _U2TXInterrupt(void)
{
 if (*str != 0)
  {
   U2TXREG = *str++;
   while(U2STAbits.UTXBF == 1);
  }
}

void main()
{

  UART2Init();
  _U2TXIF = 0;
  _U2TXIE = 1;
  U2TXREG = *(--str);

  while (1)
  {
  }
}
```

Программа отправляет строку байтов `str` в канал 2 модуля последовательного обмена микроконтроллера PIC24, для чего используется прерывание передатчика с функцией-обработчиком `_U2TXInterrupt`. Для управления передачей данных по линии CTS в функции инициализации `UART2Init` необходим оператор

```
U2MODEbits.UEN1 = 1;
```

Сигнал управления поступает на вывод U2CTS, который функционально эквивалентен CTS, с бита 0 порта P1 принимающего устройства на микрокон-

троллере 8052. Передача строки байтов начинается при активном низком уровне на выводе U2CTS.

Исходный текст программы, управляющей приемом данных в устройстве DD2, представлен в листинге 2.8.

Листинг 2.8. Программа, управляющая приемом данных по интерфейсу RS-232 с синхронизацией передатчика

```
#include <REG52.H>

sbit RTS = P1^0;
char Temp;

void INT0_ISR(void) interrupt 0
{
  RTS = 0;
}

void Serial_ISR(void) interrupt 4
{
 if (RI)
  {
   RI = 0;
   Temp = SBUF;
   if (Temp == 0x20)
     RTS = 1;
  }
}

void main(void)
{
  P1 = 0x0;
  RTS = 1;

  SCON = 0x50;
  TMOD = 0x21;
  TH1 = 0x0FD;
```

```
  TR1 = 1;
  TI = 1;

  EX0 = 1;
  IT0 = 1;
  ES = 1;
  EA = 1;
  while(1);
 }
```

В этой программе присутствуют два обработчика прерывания — INT0 и последовательного порта. Функция-обработчик `INT0_ISR` прерывания INT0 сбрасывает линию RTS (вывод P1.0 микроконтроллера 8052), изначально установленную в высокий уровень, в низкий уровень, разрешая тем самым работу передатчика на микроконтроллере PIC24. Само прерывание INT0 инициируется по перепаду 1→0 сигнала на соответствующем входе.

Прием байтов данных осуществляется в обработчике прерывания последовательного порта `Serial_ISR`. Прием осуществляется до обнаружения первого пробела в последовательности байтов, после чего на линии RTS устанавливается высокий уровень напряжения, который запрещает дальнейшую передачу данных с устройства DD1.

В рассмотренных примерах для обмена данными мы использовали системы на базе микроконтроллеров. В качестве одного из устройств, участвующих в обмене данными, нередко выступает последовательный порт персонального компьютера. Программирование последовательного порта ПК для обмена данными с внешними устройствами рассмотрено в *гл. 4* и *5* применительно к операционным системам Windows и Linux.

Протокол RS-232 имеет ряд существенных ограничений по скорости обмена, протяженности сигнальных линий и помехозащищенности, и, кроме того, он позволяет одновременно работать только двум устройствам. Он не подходит для разветвленных и протяженных сетей обмена данными с высокой помехозащищенностью. В результате серьезных усилий протокол RS-232 был усовершенствован и развился в такие протоколы, как RS-422 и RS-485. Протокол RS-485 рассмотрим более подробно. Он очень широко используется при разработке промышленных сетей обмена данными и управления.

2.4. Интерфейс RS-485

При использовании интерфейса RS-232 возникает серьезная проблема — низкая помехозащищенность. Сигналы интерфейса RS-232 привязаны к нулевому уровню напряжения, поэтому помехи по нулевому проводу могут вызывать помехи, искажающие полезный сигнал. Зона нечувствительности сигналов интерфейса RS-232 составляет ±3 В, этого может оказаться недостаточно для защиты от мгновенных перепадов напряжений по цифровым линиям. Если схема работает со стандартными уровнями сигналов ±5 В, что соответствует TTL-логике, то сигналы тактирования цифровых схем могут вызывать появление ложных сигналов на земляной шине в том случае, если шины питания расположены рядом. Такие сигналы могут интерпретироваться программным обеспечением, как биты посылки, что и приведет к ошибкам.

Подобные ограничения были устранены в протоколе RS-485, который был разработан как усовершенствованный вариант протокола RS-232. Сигналы интерфейса RS-485 не привязаны к нулевому уровню, а являются дифференциальными, что значительно ослабляет синфазные помехи. Интерфейс RS-485 очень широко применяется при создании сетей промышленной автоматики и для надежной передачи данных между удаленными объектами. Протокол RS-485 лежит в основе популярных промышленных сетей управления Profibus и Modbus.

Сеть, построенная на интерфейсе RS-485, представляет собой приемопередатчики, соединенные при помощи витой пары. В основе интерфейса RS-485 лежит принцип *дифференциальной* (балансной) передачи данных. Суть его заключается в передаче одного сигнала по двум проводам. Причем по одному проводу (условно A) идет оригинальный сигнал, а по другому (условно B) — его инверсная копия. Другими словами, если на одном проводе "1", то на другом — "0", и наоборот. Таким образом, между двумя проводами витой пары всегда есть разность потенциалов: при "1" она положительная, при "0" — отрицательная.

Принцип передачи сигналов по интерфейсу RS-485 проиллюстрирован на рис. 2.14.

Сигналы в линиях A и B могут принимать неотрицательные значения и являются инверсными по отношению друг к другу, а дифференциальный сигнал представляет собой их разность, т. е. уровень напряжения сигнала в линии A вычитается из уровня напряжения в линии B. Таким образом, получается чистый дифференциальный сигнал, который не зависит от уров-

ней синфазных составляющих в линиях A и B, поскольку при вычитании разность таких постоянных сигналов дает 0.

Такой метод формирования сигнала отличается от того, который используется в RS-232 и базируется на уровне напряжения относительно общего провода схемы. Любая помеха с достаточно высоким уровнем напряжения по общему проводу приведет к искажению сигнала в интерфейсе RS-232.

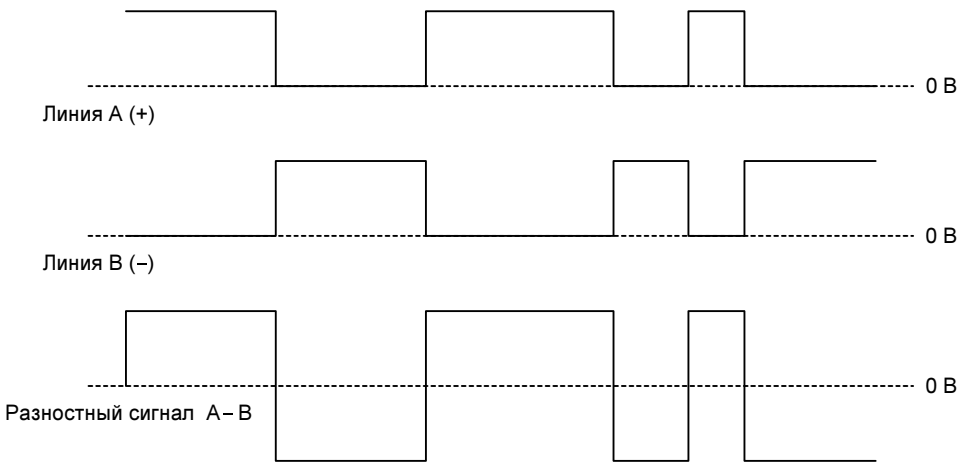


Рис. 2.14. Принцип передачи сигнала по интерфейсу RS-485

Использование дифференциальных сигналов в интерфейсе RS-485 позволяет осуществлять взаимодействие между узлами, находящимися на значительных расстояниях друг от друга (до 1200 метров), что намного больше, чем позволяет протокол RS-232 (до 15 м). Максимальная скорость обмена данными по протоколу RS-485 равна 10 Мбит/с, хотя современные микросхемы драйверов интерфейса RS-485 позволяют работать со скоростями до 35 Мбит/с.

Протокол RS-485 позволяет строить разветвленные сети передачи данных при высокой надежности, что делает его весьма популярным при создании сетей промышленной и лабораторной автоматики. Это единственный протокол, позволяющий осуществлять взаимодействие между несколькими передатчиками и приемниками, находящимися в одной и той же сети. При использовании стандартных приемников RS-485 с входным сопротивлением 12 кОм можно создать сеть, состоящую из 32 устройств. В настоящее время выпускаются драйверы интерфейса RS-485 с входным сопротивлени-

ем до 100 кОм и выше, что позволяет подключать к одной сети до 256 устройств.

Расширить сети на базе протокола RS-485 можно и за счет применения усилителей (репитеров) сигналов, что дает возможность включать в сеть несколько сотен узлов и увеличить расстояние, на котором они взаимодействуют, до нескольких километров. Еще одним существенным преимуществом интерфейса RS-485 является простота его реализации, сопоставимая с протоколом RS-232. По этим причинам интерфейс RS-485 приобрел очень широкую популярность при создании сетей на базе ПК, микроконтроллеров и интеллектуальных измерительных систем.

Топологию сети на базе протокола RS-485 иллюстрирует рис. 2.15.

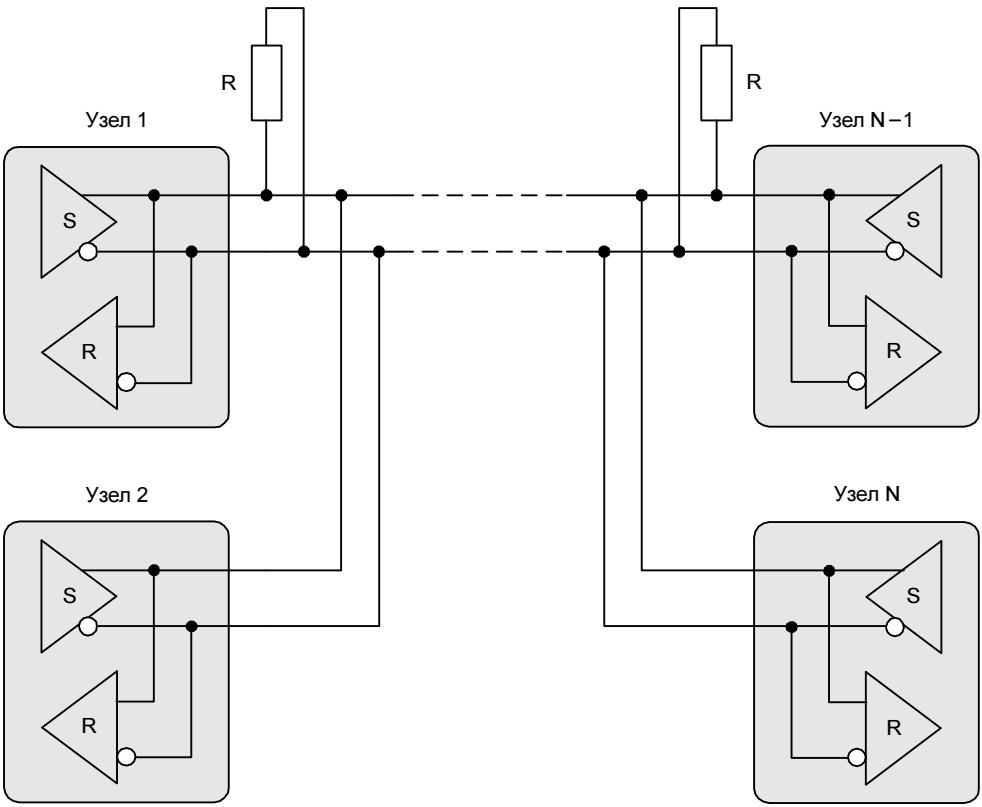


Рис. 2.15. Топология сети на базе протокола RS-485

На этом рисунке показана сеть на базе RS-485, состоящая из N узлов. Для обеспечения высокой скорости передачи данных и снижения уровня отраженных сигналов в длинных линиях на обоих концах линии нужно устанавливать терминирующие резисторы, сопротивление которых может находиться в пределах 100—120 Ом. Топология сети разработана по принципу "общей шины". Реализация сети в виде "звезды" не допускается из-за невозможности правильного терминирования линий, что приводит к существенному падению производительности. Сигнальная линия должна выполняться "витой парой", к которой присоединяются все устройства шины. При этом расстояние от линии до микросхем интерфейса RS-485 должно быть по возможности минимальным.

Интерфейс RS-485, в отличие от RS-232, работает в полудуплексном режиме. Прием и передача идут по одной паре проводов с разделением по времени. В сети может быть много передатчиков, которые отключаются в режиме приема. По умолчанию все передатчики на шине RS-485 находятся в высокоимпедансном состоянии. С помощью программного обеспечения более высокого уровня такой сети один из узлов шины назначается "ведущим": он управляет запросами и инициирует команды. Все остальные узлы работают в режиме "ведомого" и получают запросы и команды "ведущего". Взаимодействие с "ведущим" одновременно могут осуществлять несколько узлов, при этом синхронизация такого взаимодействия выполняется определенными уровнями программного обеспечения конкретной сети. Протокол RS-485 лежит в основе таких популярных сетевых протоколов, как Profibus и Modbus, которые очень широко используются в промышленности.

2.5. Примеры аппаратно-программной реализации протокола RS-485

Рассмотрим несколько примеров практической реализации протокола RS-485 при построении систем обмена данными. В качестве драйверов, формирующих уровни сигналов интерфейса, будем использовать популярные микросхемы MAX487. В первом проекте выполняется обмен данными между двумя системами на базе микроконтроллеров 8052 (рис. 2.16).

Проанализируем аппаратную часть интерфейса. В качестве драйверов интерфейса RS-485 здесь используются микросхемы MAX487.

Микросхема имеет следующие обозначения выводов:

- DI (Driver Input) — цифровой вход передатчика;
- RO (Receiver Output) — цифровой выход приемника;
- DE (Driver Enable) — разрешение работы передатчика;
- RE (Receiver Enable) — разрешение работы приемника;
- A — прямой дифференциальный вход-выход;
- B — инверсный дифференциальный вход-выход.

Функциональная схема драйвера линии MAX487 является классической для интерфейса RS-485 (рис. 2.17).

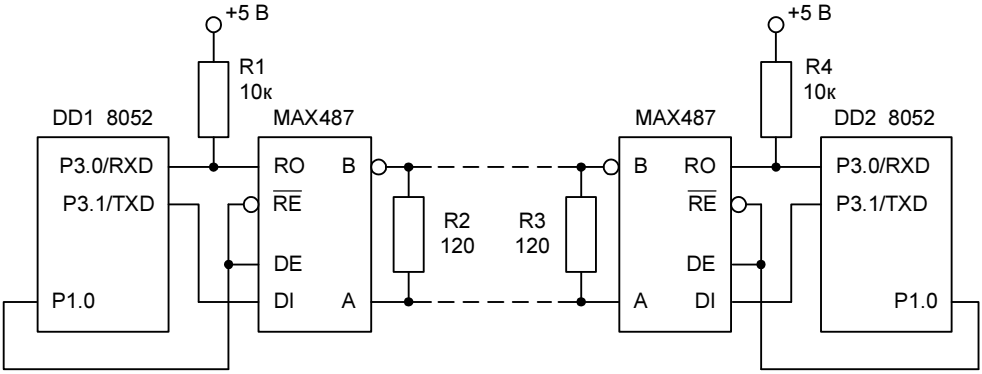


Рис. 2.16. Аппаратная реализация интерфейса RS-485 между двумя устройствами

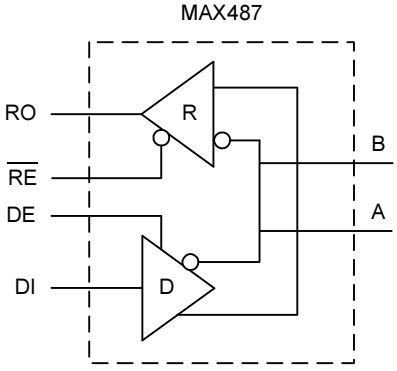


Рис. 2.17. Функциональная схема драйвера MAX487

Выход приемника RO подключается к входной линии RXD асинхронного приемопередатчика микроконтроллера 8052 (микросхема DD1), а вход передатчика DI — к линии TXD микроконтроллера (см. рис. 2.16). Поскольку на дифференциальной стороне приемник и передатчик соединены, то во время приема отключается передатчик, а во время передачи — приемник. Для этого служат управляющие входы разрешения приемника (RE) и передатчика (DE). Поскольку вход RE инверсный, то его можно соединить с DE и переключать приемник и передатчик единственным сигналом с порта P1.0 микроконтроллера 8052 (0 — прием, 1 — передача).

При работе микросхемы MAX487 на передачу выход приемника RO переводится в третье состояние, и вывод RXD микроконтроллера будет находиться в "обрыве". В момент передачи на входе приемника UART может оказаться не уровень напряжения стопового бита (логическая "1"), а произвольное напряжение, которое может быть интерпретировано как входной сигнал. Чтобы этого избежать, можно присоединить вход RXD к источнику питания схемы через резистор сопротивлением в несколько килоом.

В этой схеме данные передаются от микроконтроллера DD1 микроконтроллеру DD2. Исходный текст программы передатчика на языке ассемблера приведен в листинге 2.9.

Листинг 2.9. Программа передачи данных по интерфейсу RS-485

```
     org   0h
     jmp   start
Serial_ISR:
     org   23h
     jbc   TI, TRN
     reti
TRN:
     clr   A
     movc  A, @A+DPTR
     cjne  A, #0, continue
     reti
continue:
     mov  SBUF, A
     inc  DPTR
     reti
```

```
str1:
     DB   'Test of RS-485 Interface', 0

start:
     mov  P1, #0h
     mov  DPTR, #str1
     mov  SCON, #50h
     mov  TH1, #0FDh
     mov  TMOD, #20h
     setb TR1

    setb  EA
    setb  ES
    setb  P1.0
    mov   R5, 200
next:
    mov   R4, #250
again:
    djnz  R4, again
    djnz  R5, next
    setb  TI
    jmp   $
    end
```

Для передачи строки байтов `str1` используется функция-обработчик `Serial_ISR` прерывания последовательного порта. Направление обмена данными устанавливается командой

```
setb  P1.0
```

которая переключает буферы микросхемы MAX487 на передачу посредством установки логической "1" на входах RE—DE.

Исходный текст программы приемника приведен в листинге 2.10.

Листинг 2.10. Программа приема данных по интерфейсу RS-485

```
     org  0h
     jmp  start
```

```
Serial_ISR:
     org  23h
     jbc  RI, RCV
     reti
RCV:
     mov  A, SBUF
     reti
start:
     mov  P1, #0h
     mov  SCON, #50h
     mov  TH1, #0FDh
     mov  TMOD, #20h
     setb TR1
     setb EA
     setb ES
     clr  P1.0
     jmp  $
     end
```

Для приема байтов используется функция-обработчик `Serial_ISR` прерывания последовательного порта. Направление обмена данными устанавливается командой

```
clr  P1.0
```

которая переключает буферы микросхемы MAX487 на прием данных посредством установки логического "0" на входах RE—DE.

В следующем проекте продемонстрирована реализация интерфейса RS-485 между несколькими устройствами. Схема аппаратной части проекта показана на рис. 2.18.

В этой схеме в качестве передатчика данных используется микроконтроллер PIC24, а в качестве приемников — микроконтроллеры 8052 (DD2 и DD3). Основные элементы схемы мы уже рассматривали в предыдущем примере, поэтому останавливаться на них я не буду. Программы приема данных устройств DD2 и DD3 работают так же, как и в предыдущем примере. Исходный текст программы передачи байтов для PIC-микроконтроллера приведен в листинге 2.11.

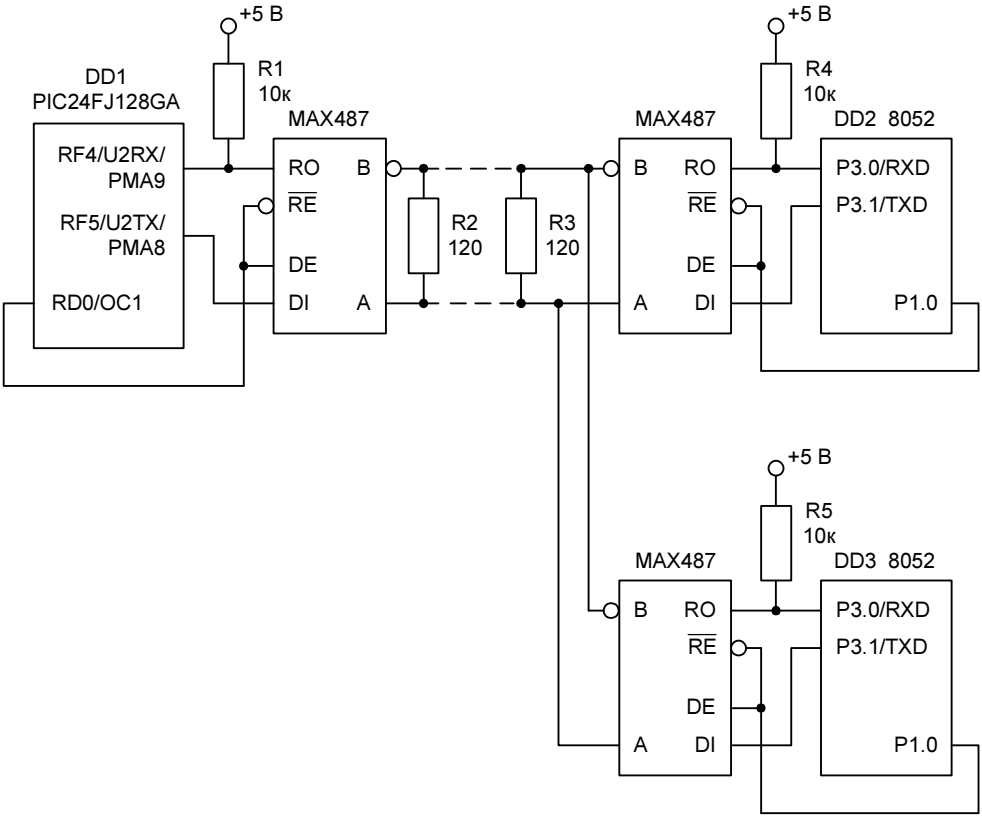


Рис. 2.18. Аппаратная реализация интерфейса RS-485 для нескольких устройств

Листинг 2.11. Программа передачи данных по интерфейсу RS-485

```
#include <p24fj128ga010.h>

_CONFIG2( FCKSM_CSDCMD & OSCIOFNC_ON & POSCMOD_HS & FNOSC_PRI )

#define SYSCLK       8000000
#define BAUDRATE2    9600
#define BAUDRATEREG  SYSCLK /8 / BAUDRATE2-1

#define UART2_TX_TRIS   TRISFbits.TRISF5
#define UART2_RX_TRIS   TRISFbits.TRISF4
```

```
void UART2Init()
{
  UART2_TX_TRIS = 0;
  UART2_RX_TRIS = 1;
  U2BRG = BAUDRATEREG;
  U2MODE = 0;
  U2MODEbits.BRGH = 1;
  U2MODEbits.UARTEN = 1;

  U2STA = 0;
  U2STAbits.UTXEN = 1;
  IFS1bits.U2RXIF = 0;
}
void  UART2PutChar(char Ch)
{

  while(U2STAbits.UTXBF == 1);
  U2TXREG = Ch;
}

void Delay(void)
{
 long del = 1000;
 while (del != 0) del--;
}

void main()
{
  char *s1 = "RS-485 Test String from PIC24";
  TRISD = 0;
  _RD0 = 1;

  UART2Init();
  Delay();
  while (*s1 != '\0')
   {
    UART2PutChar(*s1++);
```

```
    Delay();
   }
  while (1)
  {
  }
}
```

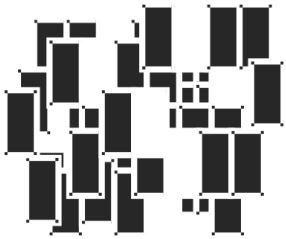
В этой программе передача строки байтов `s1` осуществляется в цикле `while (*s1 != '\0')` основной программы. Переключение драйвера MAX487 интерфейса RS-485 на передачу осуществляется установкой бита 0 порта D по команде:

```
_RD0 = 1;
```

Передачу одного байта строки выполняет функция `UART2PutChar`, а предварительную инициализацию канала 2 последовательного интерфейса микроконтроллера PIC24 выполняет функция `UART2Init`.

Для систем обмена данными интерфейса RS-485 с количеством узлов больше двух актуальной становится проблема определения адреса узла, которому направляются данные. В классическом варианте протокол RS-485 не определяет адреса устройств. Тем не менее, разработаны протоколы более высокого уровня, работающие поверх RS-485, в которых предусмотрена адресация отдельных устройств в сети на базе RS-485. В простейшем варианте адресацию устройств можно реализовать программным способом, при котором в качестве первого байта данных идет адрес устройства. Программа приемника в этом случае должна распознать свой адрес и начать обработку данных или же пропустить обработку в случае несовпадения адреса.

Глава 3



Аппаратная реализация последовательного интерфейса в компьютерных системах

Все процедуры обмена данными по последовательному интерфейсу осуществляются посредством специальных микросхем, которые получили название универсальных асинхронных приемопередатчиков (от англ. UART — Universal Asynchronous Receiver/Transmitter). Все устройства, которые выполняют последовательный обмен данными, имеют тот или иной тип микросхемы UART. Тем не менее, аппаратная часть большинства таких приемопередатчиков базируется на топологии кристалла 8250 и его более продвинутой версии 8250A. Устройства серии 8250 включают такие популярные микросхемы асинхронных приемопередатчиков, как 16450, 16550, 16650 и т. д. Практически любую из этих микросхем можно обнаружить в персональных компьютерах и других коммуникационных устройствах.

В этой главе мы рассмотрим особенности функционирования аппаратной части устройства UART и программирования обмена данными посредством асинхронного приемопередатчика на нижнем уровне.

3.1. Аппаратная архитектура UART

Асинхронный приемопередатчик является базовым звеном последовательного интерфейса, представляя собой довольно сложный аппаратно-программный модуль обмена данными. В данном разделе мы сосредоточим внимание на функционировании аппаратной части этого устройства. Основные характеристики различных микросхем UART приводятся в табл. 3.1.

Таблица 3.1. *Сравнительные характеристики различных устройств UART*

Микросхема UART	Описание
8250 (8250-B)	Устанавливался на первых моделях персональных компьютеров
16450/(8250-A)	Данное устройство полностью совместимо с UART 8250. Здесь устранены ошибки в регистре разрешения прерываний и включен ряд дополнительных возможностей
16550	В данном устройстве можно использовать внутреннюю буферизацию передаваемых и принимаемых данных посредством буфера FIFO (First In First Out), хотя эта опция работает с ошибками, из-за чего теряются отдельные символы. Обладает более высоким быстродействием по сравнению с микросхемой 16450 и позволяет работать в режиме прямого доступа к памяти (DMA — Direct Memory Access)
16550A (16550AN)	Модифицированная версия UART 16550 (устранены ошибки реализации FIFO). Предпочтительно использовать на повышенных скоростях обмена данными

Микросхемы UART очень широко используются в разработках одноплатных систем автоматизации на базе микропроцессоров и микроконтроллеров. Например, модули последовательного обмена данными всех без исключения современных микроконтроллеров реализованы на принципах работы классических UART, совместимых с 8250. В этой главе мы подробно рассмотрим принципы функционирования и методы программирования асинхронных приемопередатчиков 8250, что даст возможность достаточно глубоко изучить основные принципы реализации последовательного обмена данными и научиться создавать собственные системы передачи данных.

Устройства UART подключаются к шине сигналов/данных микропроцессорной системы. Для классической микросхемы приемопередатчика 8250A схема подключения может выглядеть так, как показано на рис. 3.1.

Приемопередатчик UART 8250A взаимодействует с системой, в которую он подключен посредством шины управления. В качестве такой шины управления может служить шина ISA/EISA/PCI компьютерных систем или специализированная шина систем на базе микроконтроллеров.

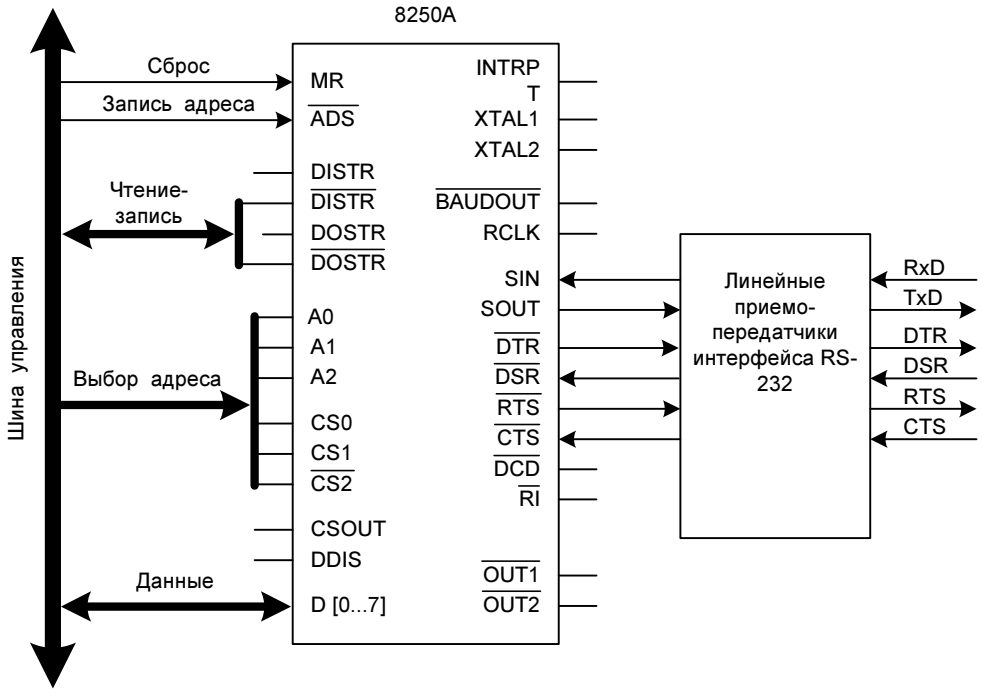


Рис. 3.1. Блок-схема подключения устройства UART 8250A

В устройстве UART можно выделить три относительно независимых функциональных модуля:

- ❒ модуль шинного интерфейса;
- ❒ модуль стандартного приемопередатчика;
- ❒ модуль управления модемом.

Модуль стандартного приемопередатчика обеспечивает передачу данных по линиям TxD и RxD с программным и аппаратным управлением потоком данных. Модуль управления модемом позволяет синхронизировать передачу-прием данных аппаратными средствами (аппаратное управление потоком данных). Модуль управления модемом используется при подключении модемов и другого телекоммуникационного оборудования, где требуется повышенная надежность передачи данных и высокое быстродействие.

Устройство UART имеет целый ряд внутренних программно доступных регистров, посредством которых микропроцессорная система может управлять обменом данными через асинхронный приемопередатчик, а также настраи-

вать параметры обмена данными. Доступ к регистрам UART осуществляется по стандартной схеме обращения к устройствам шины. Например, цикл записи данных в регистр будет выполняться в соответствии с временной диаграммой, показанной на рис. 3.2.

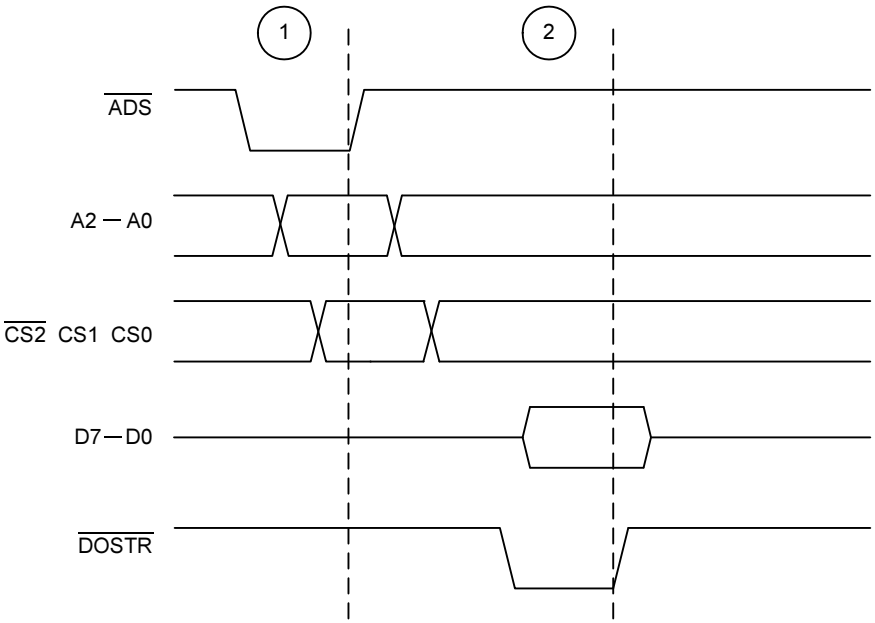


Рис. 3.2. Временная диаграмма записи данных в регистр UART, совместимый с 8250A

Это один из вариантов аппаратной реализации цикла записи байта данных в регистр UART. Здесь для записи используется инверсный строб записи DOSTR, при этом сигнал на линии прямого строба записи должен иметь низкий уровень (2). Можно разработать схему записи данных с использованием прямого строба DOSTR с активным высоким уровнем сигнала, тогда сигнал на инверсной линии должен иметь высокий уровень.

Сигнал ADS с активным низким уровнем служит для фиксации адреса регистра UART, при этом на линиях A0—A2 должен быть установлен адрес регистра (1). Можно обойтись и без строба ADS, но следует гарантировать, чтобы адрес регистра оставался неизменным в течение цикла записи. Линии nCS2, CS1—CS0 служат для выбора кристалла микросхемы и аппаратно могут быть подключены либо к соответствующим уровням напряжений, либо посредст-

вом цифровой логики запрещать/разрешать подачу сигналов чтения-записи. Аппаратная реализация шинного интерфейса в каждом конкретном случае зависит от разработчика.

По такой же схеме выполняется и операция чтения регистра UART, только здесь будут задействованы сигналы DISTR (прямой или инверсный). Кроме рассмотренных сигнальных линий микросхемы UART, 8250A и совместимые с ней имеют и другие выводы, которые должны подключаться в соответствии с технической документацией на каждое конкретное устройство. Аппаратно-программную реализацию шинного интерфейса можно выполнить на микропроцессоре или микроконтроллере, причем сделать это не так и сложно. Все сигналы микросхемы UART работают с уровнями TTL-логики, поэтому проблем по части согласования уровней сигналов на шине не возникает.

Выходные сигналы устройства UART через буферные микросхемы выводятся на внешние выводы разъема последовательного порта. В качестве линейных приемопередатчиков или, по-другому, буферов интерфейса RS-232 в большинстве случаев используется ставшая уже классической микросхема MAX232, хотя в настоящее время она активно заменяется другими устройствами, не требующими дополнительных навесных компонентов.

Все настройки устройства UART, а также прием и передача данных осуществляются посредством регистров, большая часть которых программно доступна пользовательским приложениям, а несколько регистров являются внутренними служебными регистрами и служат для выполнения специализированных операций (например, регистры сдвига приемопередатчика для передачи бита данных в линию TxD или приема бита данных по линии RxD).

Рассмотрим назначение регистров UART. Программисту доступны следующие регистры асинхронного приемопередатчика:

- регистр приема-передачи;
- регистр разрешения прерываний;
- регистр идентификации прерывания;
- регистр управления линией;
- регистр управления модемом;
- регистр состояния линии;
- регистр состояния модема;
- временный регистр.

Каждый из регистров имеет адрес, который отсчитывается от базового адреса устройства UART. В качестве базового адреса принимается адрес регистра приема-передачи. Например, для последовательного порта COM1 персонального компьютера базовый адрес в подавляющем большинстве случаев равен шестнадцатеричному значению 0x3F8, которое является адресом регистра приема-передачи устройства UART. Базовый адрес последовательного порта COM2 обычно равен 0x2F8. Если обозначить адрес базового регистра как PORT1 (такое обозначение мы будем использовать при анализе примеров программирования далее *в этой главе*), то, например, адрес регистра управления линией будет равен PORT1+3, адрес регистра состояния линии будет равен PORT1+5 и т. д.

Регистры UART имеют и специальные мнемонические обозначения. Наиболее часто мы будем использовать регистр управления линией (Line Control Register, LCR), регистр состояния линии (Line Status Register, LSR), регистр управления модемом (Modem Control Register, MCR) и регистр состояния модема (Modem Status Register, MSR). Регистры UART позволяют задавать различные режимы работы и параметры обмена данными. Для изучения практических аспектов программирования устройства UART нам понадобятся только некоторые основные моменты, касающиеся функционирования порта. Сразу замечу, что мы не будем рассматривать расширенные возможности UART, такие как работа с прерываниями и буфером FIFO, поскольку программирование таких функций, особенно в защищенных операционных системах Windows 2000/XP/Vista, требует специальных знаний в области разработки драйверов устройств и глубоких знаний самой операционной системы. Тем не менее, мы проанализируем основы программирования асинхронного приемопередатчика и научимся настраивать параметры обмена данными.

В простейшем варианте для обмена данными по последовательному порту посредством UART следует выполнить такие шаги:

- задать скорость обмена данными устройства UART;
- задать формат данных;
- выполнить передачу или прием данных.

Некоторые из этих шагов можно пропустить, например, не задавать скорость обмена данными, а использовать значение по умолчанию, принятое в системе (обычно скорость обмена устанавливается по умолчанию равной 9600 бод). Можно использовать и стандартное значение формата данных системы (8 бит, 1 стоповый бит, без контроля четности).

Скорость обмена данными задается специальным образом на основе деления тактовой частоты кристалла UART. Стандартная тактовая частота для работы микросхемы UART обычно равна 1,8432 МГц. Внутренняя частота синхронизации будет в 16 раз меньше и равна 1,8432 / 16 = 115,2 кГц. Для получения требуемой скорости обмена в оба байта делителя частоты нужно записать значение, равное частоте синхронизации в герцах деленной на скорость обмена в бодах. Например, для стандартной скорости обмена, равной 9600 бод, в регистры делителя частоты нужно записать значение 115200 / 9600 = 12 (или 0x0C в шестнадцатеричной нотации). Таким образом, в младший байт делителя частоты будет записано значение 0x0C, а в старший байт — 0x0. Если значение делителя частоты установить равным 1, то получим максимальную скорость обмена 115200 бод.

В табл. 3.2 приводятся некоторые значения делителя скорости для ряда стандартных скоростей обмена.

Таблица 3.2. *Значения делителя частоты для разных скоростей обмена*

Скорость (в бодах)	Десятичное значение	Шестнадцатеричное значение
50	2304	0900
300	384	0180
600	192	00C0
1200	96	0060
2400	48	0030
4800	24	0018
9600	12	000C
19200	6	0006
38400	3	0003
57600	2	0002
115200	1	0001

Первое, что нужно сделать для настройки скорости обмена — установить бит 7 регистра управления линией (LCR) в 1. Этот бит имеет специальное

обозначение DLAB (Divisor Latch Access Bit) и позволяет получить доступ к 16-битовому регистру делителя скорости. Он состоит из младшей 8-битовой части (DLL, Divisor Latch Low), которая доступна по базовому адресу UART, и старшей части (DLH, Divisor Latch High), доступной по базовому адресу + 1.

После установки бита DLAB нужно записать требуемое значение в оба 8-битовых регистра делителя. После установки скорости обмена следует очистить бит DLAB, иначе он будет блокировать доступ к регистру приема-передачи, что сделает обмен данными невозможным.

Следующий этап — установка формата данных. Для настройки этого параметра используются биты 0—3 регистра управления линией (LCR, базовый адрес + 3). В табл. 3.3 показаны значения этих битов для 7- и 8-битовой посылки данных с одним стоповым битом и без контроля четности.

Таблица 3.3. *Настройка формата данных в регистре управления линией*

Формат данных	**Бит 3 (контроль четности)**	**Бит 2 (количество стоповых бит)**	**Бит 1**	**Бит 0**
7 бит, 1 стоповый, без контроля четности	0	0	1	0
8 бит, 1 стоповый, без контроля четности	0	0	1	1

Таким образом, для установки 8-битовой посылки, с одним стоповым битом и без контроля четности в регистр LCR нужно записать значение 0x3.

Прием-передача данных осуществляется посредством базового регистра приемопередатчика. При этом для передачи байта нужно записать его значение в регистр приема-передачи, а при приеме данных — прочитать значение, находящееся в этом регистре. В процессе обмена данными нужно точно знать, когда очередной байт данных отправлен из регистра передачи в линию (для посылки следующего байта) или когда очередной байт находится в регистре приема. О полном завершении процесса передачи байта данных будет свидетельствовать установленный бит 6 регистра состояния линии (LSR). О наличии байта данных в регистре приема будет свидетельствовать установленный бит 0 регистра LSR. Оба бита сбрасываются устройством автома-

тически при записи очередного байта данных для передачи или при чтении полученного байта данных из регистра приема.

Все описанные ранее операции мы реализуем на практических примерах, которые приведены далее в *этой главе*.

3.2. Диагностика и настройка интерфейса RS-232

В процессе разработки приложений, для которых требуется выполнять обмен данными по интерфейсу RS-232, часто возникает необходимость настройки и проверки функционирования последовательного порта того или иного устройства на уровне аппаратных средств, т. е. асинхронного приемопередатчика (UART). Это касается как персональных компьютеров, так и специализированных устройств. Для персональных компьютеров, работающих под управлением операционных систем Windows, такую настройку и диагностику последовательного порта выполнить несложно, используя арсенал современных средств разработки (Ассемблер, Delphi, Microsoft Visual Studio и т. д.). То же самое касается и популярных операционных систем Linux, которые функционируют на многих ПК и в большинстве телекоммуникационных устройств, хотя там имеются и некоторые особенности, которые мы проанализируем позже.

В простейшем варианте тестирования и настройки COM-порта нужно соединить выводы сигнальных линий TxD и RxD на разъеме последовательного порта ПК (рис. 3.3).

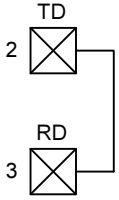


Рис. 3.3. Шлейф обратной связи для тестирования COM-порта

В этом случае можно программным способом отправлять данные по линии TxD и тут же принимать их по линии RxD одного и того же порта. Этот метод очень удобен, поскольку не требуется никакого дополнительного обору-

дования, подключенного к COM-порту. Естественно, при таком методе проверки имеются и некоторые ограничения. Например, сложно проверить работу интерфейса в реальном времени, особенно при использовании сигналов квитирования (подтверждения), как это часто требуется для реальных приложений. Тем не менее, такое тестирование позволяет довольно быстро определить работоспособность интерфейса и его функционирование при задании различных параметров обмена данными.

Если под рукой имеется демо-плата на базе микроконтроллера или микропроцессора с интегрированным на ней последовательным портом, то ее можно использовать для тестирования или настройки интерфейса последовательного обмена данными.

При разработке программной части проверки и настройки микросхемы UART интерфейса RS-232 следует учитывать особенности функционирования операционных систем, под управлением которых работает последовательный интерфейс. В следующих разделах мы проанализируем особенности настройки и тестирования асинхронного приемопередатчика последовательного интерфейса в различных операционных системах. Начнем с настройки UART в операционных системах Windows 98/Me. Эти операционные системы благодаря возможности прямого доступа к аппаратным средствам широко используются для настройки и управления периферийным оборудованием с интерфейсом RS-232, особенно в лабораторных и промышленных системах сбора и обработки данных. Это не в последнюю очередь связано и с тем, что фирма Microsoft выпустила программное обеспечение (Virtual PC 2004/2007) для создания различных виртуальных операционных систем, включая Windows 98/Me, работающих в установленных на ПК системах Windows XP/2003/Vista. Самое существенное это то, что при работе в таких виртуальных системах пользователь получает прямой доступ к аппаратным ресурсам параллельного и последовательного портов без сложной и трудоемкой процедуры написания драйверов ядра, а это открывает широкие возможности по адаптации и настройке оборудования относительно простыми средствами.

3.2.1. Настройка и тестирование UART в операционных системах Windows 98/Me

Для настройки/тестирования устройства UART в операционных системах Windows 98/Me мы будем использовать две наиболее популярные программные среды разработки — Microsoft Visual Studio и Delphi. В данном случае

мы будем использовать Delphi 7 и Visual C++ 6 пакета Microsoft Visual Studio 6, хотя программы можно компилировать и в других версиях этих программных продуктов.

Для настройки/тестирования асинхронного приемопередатчика COM-порта ПК можно соединить выводы TxD и RxD последовательного порта или же подключить к последовательному порту устройство, которое могло бы принимать данные из порта или пересылать их в порт. Программное обеспечение пользователя может либо читать данные из последовательного порта, отображая их на экране консоли, либо отправлять данные в порт присоединенному устройству.

Конечно, использование замкнутого шлейфа TxD—RxD ("заглушки") позволяет при минимальных усилиях проверить и настроить большинство параметров обмена данными, а также проверить работу самого порта. Тем не менее, следует учитывать, что в реальных системах должны выдерживаться определенные временные зависимости между сигналами интерфейса приемного и передающего устройства, поэтому окончательную настройку процедуры обмена данными желательно все же выполнять с реальным оборудованием, подключенным к последовательному порту.

Тем не менее, использование замкнутого шлейфа дает программисту возможность без установки оборудования достаточно хорошо изучить принципы обмена данными. Это все, что касается аппаратных подключений. Теперь посмотрим, что нужно для программной части. Во-первых, на персональном компьютере нужно проинсталлировать одну из операционных систем Windows 98/Me. Поскольку в настоящее время многие пользователи уже имеют установленную версию одной из операционных систем Windows 2000/XP/Vista, то можно пойти и другим путем. В этом случае лучше всего использовать бесплатно предоставляемую Microsoft программную среду Virtual PC 2007, с помощью которой можно без особого труда установить "виртуальную" полнофункциональную Windows 98/Me. Естественно, нужно иметь дистрибутив Windows 98/Me. Установка "виртуальной машины" довольно проста и достаточно хорошо описана в документации Microsoft.

Во-вторых, в операционной системе Windows 98/Me нужно установить предпочтительную среду разработки (Delphi или Visual Studio). Наконец, желательно иметь установленную программу терминального доступа по последовательному порту — это не проблема, поскольку имеется великое множество таких программ. В принципе, можно обойтись и без такой программы, но ее использование существенно упрощает тестирование/настройку.

Перейдем к разработке программного обеспечения для тестирования/настройки последовательного порта, вернее, его аппаратной части — асинхронного приемопередатчика (UART). Начнем с примеров обмена данными, разработанных в среде Delphi 7. Первое приложение будет отправлять строку байтов терминальной программе, запущенной на ПК при соединенных сигнальных линиях TxD и RxD. Эту схему мы будем часто использовать при настройке/тестировании последовательного порта (рис. 3.4).

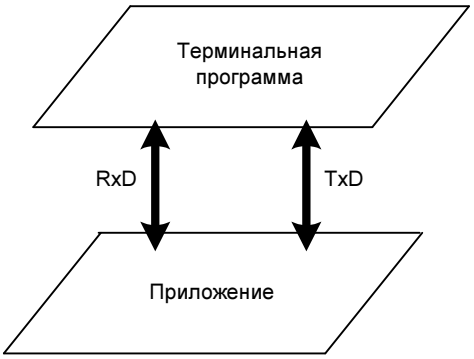


Рис. 3.4. Схема взаимодействия приложения и терминальной программы

Вместо терминальной программы для настройки порта можно использовать другое приложение. В этом случае одна пользовательская программа отправляет данные, а другая — их принимает.

Вид окна конструктора приложения показан на рис. 3.5.

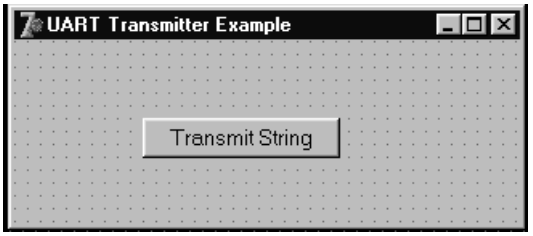


Рис. 3.5. Вид окна конструктора приложения

Приложение работает предельно просто: при нажатии кнопки **Transmit String** терминальной программе отправляется строка байтов. Вся работа про-

граммы выполняется функцией-обработчиком нажатия кнопки. Исходный текст программы приведен в листинге 3.1.

Листинг 3.1. Передача строки байтов посредством UART

```
 unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  s1: PChar;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
const
 PORT1 = $3F8;
 LSR = $3FD;
```

```
begin
 s1:= 'Hello,Dumb Terminal! ';
 asm
  mov  ESI, dword ptr s1
@next_ch:
  mov AL, byte ptr [ESI]
  mov DX, 3F8h

  cmp AL, 0
  je  @quit
  out DX, AL

  mov DX, 3FDh
@wait_trm:
  in  AL, DX
  and AL, 40h
  cmp AL, 40h
  jne @wait_trm
  mov BL, 100
@delay:
  dec BL
  jnz @delay
  inc ESI
  jmp @next_ch
@quit:
 end
end;

end.
```

Программа передает строку байтов (переменная `s1`) посредством прямого программирования регистров устройства UART в процедуре `Button1Click`. Процедура написана на встроенном ассемблере — это позволяет лучше почувствовать работу аппаратуры, хотя при желании можно ее переписать с помощью операторов высокого уровня. Адрес строки помещается в ре-

гистр ESI, посредством которого осуществляется доступ к отдельным ее элементам:

```
mov  ESI, dword ptr s1
```

В каждой итерации байт помещается в регистр AL, где он проверяется на равенство нулю. В Delphi строка типа PChar заканчивается нулем, что и будет служить признаком окончания передачи. Затем, если символ в регистре AL не равен 0, то командой out он выводится в базовый порт приемопередатчика, который содержится в регистре DX (в данном случае выбран стандартный адрес, равный 0x3F8). Описанные действия выполняются следующей группой команд:

```
mov AL, byte ptr [ESI]
mov DX, 3F8h
. . .
cmp AL, 0
je  @quit
out DX, AL
```

Команда out записывает байт данных из регистра AL в регистр приемопередатчика UART, который является базовым регистром COM-порта. Байт данных из регистра передатчика записывается в регистр сдвига (это внутренняя операция UART), откуда синхронно с тактовыми импульсами синхронизации биты выдвигаются в линию TxD. Процесс передачи байта считается полностью законченным, когда регистр приемопередатчика и регистр сдвига будут пустыми, о чем свидетельствует установленный бит 6 регистра состояния линии LSR (PORT1 + 5). Установленный бит 5 этого регистра свидетельствует о том, что регистр передатчика пуст, но регистр сдвига продолжает передачу битов на линию TxD.

Следующий байт можно передавать при любом установленном бите 5 или 6. В нашей программе проверяется состояние бита 6 регистра LSR:

```
  mov DX, 3FDh
@wait_trm:
  in  AL, DX
  and AL, 40h
  cmp AL, 40h
  jne @wait_trm
```

После передачи байта (регистр передачи и регистр сдвига пусты) в программе выполняется некоторая задержка с помощью команд

```
@delay:
  dec BL
  jnz @delay
```

В принципе, в реальном приложении такая задержка может и не понадобиться, но в данном случае программа-терминал и программа-передатчик работают с одним и тем же портом, поэтому между передачей символов делается определенная задержка, что позволяет избежать переполнения (особенно если FIFO-буфер отключен). В любом случае можно поэкспериментировать с настройкой режима передачи с задержкой и без нее.

По окончании передачи байта указатель смещается к следующему символу, и цикл повторяется до тех пор, пока не будет обнаружен признак конца строки (нулевой байт):

```
inc ESI
jmp @next_ch
```

Замечу, что в этой программе не выполнялась настройка параметров передачи. Эти параметры берутся по умолчанию из настроек системы или настроек терминальной программы.

Следующий пример показывает, как можно принимать байты данных по последовательному порту посредством прямого доступа к регистрам UART. В этом примере символы, вводимые в окне терминальной программы, будут отображаться в окне приложения. Как и в предыдущем примере, для тестирования следует соединить линии TxD и RxD.

Вид конструктора окна приложения показан на рис. 3.6.

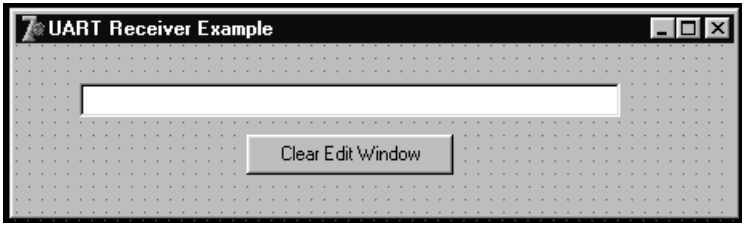


Рис. 3.6. Вид окна конструктора приложения

Программа чтения данных с последовательного порта сложнее, чем та, которую мы анализировали в предыдущем примере. Для ожидания приема байта в программе нужно создать отдельный поток, который будет непрерывно анализировать бит 0 регистра состояния линии (`LSR`) на равенство 1. Как только бит 0 будет установлен в 1, это будет означать, что данные из входного регистра сдвига поступили в регистр приемника (`PORT1`) и могут быть прочитаны. Прочитанный байт отображается в окне редактирования, а курсор смещается вправо в ожидании нового символа. Очистка буфера окна осуществляется при нажатии кнопки **Clear Edit Window** (см. рис. 3.6).

Бит 0 регистра `LSR` при чтении регистра приемника (`PORT1`) автоматически сбрасывается, поэтому в программе нет необходимости делать это вручную. Исходный текст программы приведен далее в листинге 3.2.

Листинг 3.2. Прием байтов из последовательного порта посредством UART

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    procedure FormActivate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure UART_init;

  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
 type
  Thread1 = class(TThread)
  protected
    procedure Execute; override;
    procedure UpdateEdit;
 end;

const
 PORT1 = $3F8;
 BRH   = PORT1 + 1;
 FIFO  = PORT1 + 2;
 LCR   = PORT1 + 3;
 MCR   = PORT1 + 4;
 LSR   = PORT1 + 5;

var
  myThread: Thread1;
  Form1: TForm1;
  c1: Byte;
implementation

{$R *.dfm}
procedure TForm1.UART_init;
begin
 asm
  mov  DX, BRH
  xor  AL, AL
  out  DX, AL

  mov  DX, LCR
  mov  AL, 80h
  out  DX, AL

  mov  DX, PORT1
  mov  AL, 0Ch
  out  DX, AL
```

```
  mov  DX, BRH
  xor  AL, AL
  out  DX, AL

  mov  DX, LCR
  mov  AL, 3h
  out  DX, AL

  mov  DX, FIFO
  xor  AL, AL
  out  DX, AL

  mov  DX, MCR
  out  DX, AL
 end;
end;

procedure Thread1.Execute;
begin
 while True do
  begin
   asm
     mov DX, LSR
    @again:
     in  AL, DX
     and AL, 1h
     cmp AL, 1
     jne @again
     mov DX, PORT1
     in  AL, DX
     mov c1, AL
   end;
     Synchronize(UpdateEdit);
  end;
end;
```

```
procedure Thread1.UpdateEdit;
begin
 Form1.Edit1.Text:= Form1.Edit1.Text + Char(c1);
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
 UART_init();
 Edit1.Text:= ' ';
 myThread:= Thread1.Create(False);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 Edit1.Text:= ' ';
end;

end.
```

Основная часть программы реализована на встроенном ассемблере. В отличие от предыдущей, в этой программе выполняется инициализация асинхронного приемопередатчика, реализованная в функции `UART_init`. Поскольку мы не собираемся выполнять какие-либо действия при возникновении прерывания UART, то в данном приложении прерывания асинхронного приемопередатчика запрещаются:

```
mov  DX, BRH
xor  AL, AL
out  DX, AL
```

Здесь в регистр разрешения прерываний (обозначен как `BRH`) записывается 0. Этот же регистр используется для записи старшего байта делителя частоты при установке скорости обмена. Далее нам следует установить скорость обмена данными. Это можно сделать только при установке бита 7 регистра управления линией (`LCR`) в 1:

```
mov  DX, LCR
mov  AL, 80h
out  DX, AL
```

Бит 7 называют еще DLAB (Divisor Latch Access Bit). Теперь при установленном бите DLAB запишем в старший и младший байты делителя частоты требуемые значения. В качестве младшего байта делителя используется базовый регистр (`PORT1`), в который записывается значение 0x0C, что соответствует скорости обмена 9600 бод. В старший байт делителя частоты, в качестве которого используется регистр `BRH`, запишем 0. Описанные выше установки реализованы командами

```
mov  DX, PORT1
mov  AL, 0Ch
out  DX, AL

mov  DX, BRH
xor  AL, AL
out  DX, AL
```

Следующая настройка, которую нужно выполнить — установка формата принимаемых данных. Выберем стандартный формат: 8 бит, 1 стоповый, без проверки четности. Эти настройки выполняются путем установки первых двух младших бит в регистре управления линией `LCR`. Одновременно с этим нам нужно сбросить бит DLAB, иначе регистр `PORT1` будет заблокирован для приема. Описанные действия выполняет следующая группа команд:

```
mov  DX, LCR
mov  AL, 3h
out  DX, AL
```

В программе не будет использоваться буфер FIFO и аппаратный контроль обмена данными, поэтому регистр `FIFO` и регистр управления модемом `MCR` будут заблокированы:

```
mov  DX, FIFO
xor  AL, AL
out  DX, AL

mov  DX, MCR
out  DX, AL
```

Собственно, на этом инициализация устройства UART заканчивается.

Прием данных выполняется, как уже было сказано, в отдельном потоке (назовем его `myThread`), для которого мы переопределим функцию потока `Execute`, принимающую байты данных. Признаком появления байта данных в

буфере приема является установленный в 1 бит 0 регистра состояния линии LSR, поэтому функция потока непрерывно анализирует состояние этого бита:

```
mov DX,  LSR
@again:
 in  AL, DX
 and AL, 1h
 cmp AL, 1
 jne @again
```

Цикл повторяется по метке @again до тех пор, пока бит 0 регистра LSR не станет равным 1. После этого байт из регистра приемника можно прочитать в регистр AL и сохранить его в переменной c1 для дальнейшего отображения в окне редактирования Edit1:

```
mov DX, PORT1
in  AL, DX
mov c1, AL
```

Полученное в переменной c1 значение символа нужно отобразить в окне Edit1. Для этого следует использовать метод Synchronize для функции UpdateEdit1, с помощью которой выполняется визуальное отображение символ за символом полученных данных.

Я не буду останавливаться на программных деталях реализации потока в приложении на Delphi — все это описано в документации на данный программный продукт.

Для тестирования приема символов в устройстве UART нужно запустить наше приложение-приемник и терминальную программу или же вместо терминальной программы использовать программу из предыдущего примера для передачи строки данных.

Можно обойтись вообще одной программой тестирования/настройки, если объединить программный код предыдущих примеров в одном приложении. В этом случае окно конструктора такого приложения будет выглядеть так, как показано на рис. 3.7.

В этой программе передача строки на линию TxD осуществляется при нажатии кнопки **Send String**, в результате чего строка текста появляется в окне редактирования. Кнопка **Clear Edit Window** служит для очистки содержимого окна редактирования. Напомню, что наше приложение будет работать при соединении линий TxD и RxD последовательного порта.

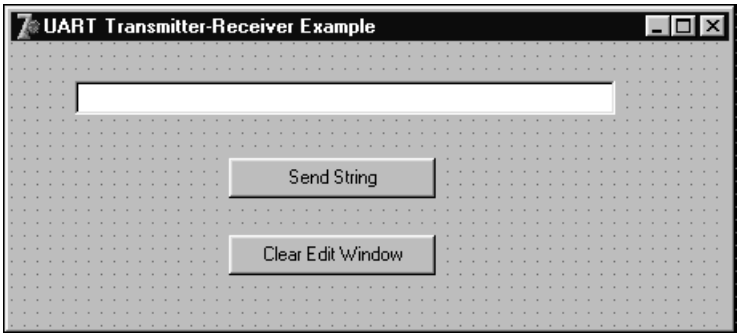


Рис. 3.7. Вид окна конструктора приложения

Исходный текст приложения приведен далее в листинге 3.3.

Листинг 3.3. Передача-прием данных через устройство UART в одной программе

```
unit Unit1;


interface


uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls;


type
  TForm1 = class(TForm)
    Edit1: TEdit;
    clButton: TButton;
    sButton: TButton;
    procedure FormActivate(Sender: TObject);
    procedure clButtonClick(Sender: TObject);
    procedure UART_init;
    procedure sButtonClick(Sender: TObject);


  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;

 type
  Thread1 = class(TThread)
  protected
    procedure Execute; override;
    procedure UpdateEdit;
 end;

const
 PORT1 = $3F8;
 BRH   = PORT1 + 1;
 FIFO  = PORT1 + 2;
 LCR   = PORT1 + 3;
 MCR   = PORT1 + 4;
 LSR   = PORT1 + 5;

var
  myThread: Thread1;
  Form1: TForm1;
  s1: PChar;
  c1: Byte;
implementation

{$R *.dfm}
procedure TForm1.UART_init;
begin
 asm
  mov  DX, BRH
  xor  AL, AL
  out  DX, AL

  mov  DX, LCR
  mov  AL, 80h
  out  DX, AL
```

```
  mov  DX, PORT1
  mov  AL, 0Ch
  out  DX, AL

  mov  DX, BRH
  xor  AL, AL
  out  DX, AL

  mov  DX, LCR
  mov  AL, 3h
  out  DX, AL

  mov  DX, FIFO
  xor  AL, AL
  out  DX, AL

  mov  DX, MCR
  out  DX, AL
 end;
end;

procedure Thread1.Execute;
begin
 while True do
  begin
   asm
     mov DX, LSR
    @again:
     in  AL, DX
     and AL, 1h
     cmp AL, 1
     jne @again
     mov DX, PORT1
     in  AL, DX
     mov c1, AL
   end;
```

```
    Synchronize(UpdateEdit);
  end;
end;

procedure Thread1.UpdateEdit;
begin
 Form1.Edit1.Text:= Form1.Edit1.Text + Char(c1);
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
 UART_init();
 Edit1.Text:= ' ';
 myThread:= Thread1.Create(False);
end;

procedure TForm1.clButtonClick(Sender: TObject);
begin
 Edit1.Text:= ' ';
end;

procedure TForm1.sButtonClick(Sender: TObject);
const
 PORT1 = $3F8;
 LSR = $3FD;
begin
 s1:= 'Loopback Test succeeded! ';
 asm
  mov  ESI, dword ptr s1
@next_ch:
  mov AL, byte ptr [ESI]
  mov DX, 3F8h

  cmp AL, 0
  je  @quit
  out DX, AL
```

```
  mov DX, 3FDh
@wait_trm:
  in  AL, DX
  and AL, 40h
  cmp AL, 40h
  jne @wait_trm
  mov BL, 100
@delay:
  dec BL
  jnz @delay
  inc ESI
  jmp @next_ch
@quit:
 end

end;

end.
```

Исходный текст программы был описан ранее, поэтому останавливаться на нем подробно я не буду. Передача строки текста (переменная `s1`) в последовательный порт осуществляется при нажатии кнопки `sButton` (**Send String** на рис. 3.7). Для приема данных по линии RxD устройства UART запускается отдельный поток, функция `Execute` которого ожидает поступления байта данных, отображаемого затем в окне редактирования `Edit1`.

Программу можно усовершенствовать, если добавить в нее еще одно окно редактирования (компонент `TEdit`), в котором набирать строку текста для передачи через устройство UART. Полученный по линии RxD текст будет отображаться во втором окне редактирования. При желании читатели могут выполнить такую модификацию программы в качестве упражнения.

Приведенные выше примеры можно реализовать на C++ в среде Microsoft Visual C++ 6. Для разработки тестовых приложений в мастере проектов следует выбрать 32-битовое консольное приложение Windows без дополнительных файлов (empty project), в которое затем можно будет включить наш файл с исходным текстом программы.

Исходный текст C++-файла для программы передачи строки байтов в последовательный порт показан в листинге 3.4.

Листинг 3.4. Программа передачи данных через UART на C++

```
#include <stdio.h>
#include <conio.h>

#define PORT1 0x3F8
#define LSR   PORT1 + 5

void main(void)
{

 char *str = "Hi, Cruel Terminal!";
 int c;

 printf("\nSample Transmitter Comm's Program.  \n");
 while (*str != 0)
  {
    _outp(PORT1, *str++);
    do
    {
     c = _inp(LSR) & 0x40;
    } while (c == 0);
  }
}
```

Проанализируем исходный текст программы. Здесь для обращения к регистрам UART используются стандартные библиотечные функции Visual C++ `_outp` и `_inp`, которые определены в файле заголовка `conio.h`. Для работы такого приложения следует запустить программу-терминал, ожидающую приема байтов по линии RxD. Передача байтов данных строки (переменная `str`) выполняется в цикле `while`, в начале которого проверяется признак конца строки (нулевой байт). Если конец строки не достигнут, то очередной байт отправляется на линию TxD устройства UART с помощью оператора

```
_outp(PORT1, *str++);
```

Здесь же выполняется продвижение указателя `str` к следующему байту строки. Следующий байт будет передаваться только при полном завершении передачи предыдущего, о чем будет свидетельствовать установленный бит 6 в регистре состояния линии (`LSR`). Проверка этого бита будет осуществляться в цикле `do…while`. Затем цикл повторяется. В этом приложении мы не выполняем настройку параметров обмена данными, используя установленные по умолчанию в системе.

Приложение, выполняющее прием байтов данных с терминальной программы, будет включать C++-файл с исходным текстом, представленным в листинге 3.5.

Листинг 3.5. Программа на C++ для приема данных через UART

```
#include <stdio.h>
#include <conio.h>

#define PORT1 0x3F8
#define LSR   PORT1 + 5

void main(void)
{
 char buf[128];
 char *pbuf = buf;
 int c;

 printf("\nWaiting for serial data to be received... \n");

 while (1)
 {
    while ((c = _inp(LSR) & 0x1) != 0x1);
   *pbuf = _inp(PORT1);
     printf("%c", *pbuf++);
 };
}
```

В этом приложении используются настройки последовательного порта, принятые в системе. Программа в цикле `while` проверяет состояние бита 0 реги-

стра состояния линии LSR и затем принимает байты данных с линии RxD в буфер данных buf. Принятый символ отображается на экране консоли, после чего указатель адреса следующего байта инкрементируется.

Для проверки/настройки UART в одной программе можно разработать приложение, которое будет отправлять байты по линии TxD и принимать их по RxD — в этом случае можно обойтись без терминальной программы. Далее представлен исходный текст C++-файла такого приложения (листинг 3.6).

Листинг 3.6. Проверка передачи/приема данных в одном приложении

```
#include <stdio.h>
#include <conio.h>

#define PORT1 0x3F8
#define BRH   PORT1 + 1
#define FIFO  PORT1 + 2
#define LCR   PORT1 + 3
#define MCR   PORT1 + 4
#define LSR   PORT1 + 5

void main(void)
{

 int c;
 int ch;
 _outp(BR, 0);
 _outp(LCR, 0x80);
 _outp(PORT1, 0x0C);
 _outp(BR, 0x00);
 _outp(LCR, 0x03);
 _outp(FIFO, 0x0);
 _outp(MCR, 0x0);
  printf("\nSample Comm's Program. Press ESC to quit \n");
   do {
       c = _inp(LSR) & 0x1;
      if (c)
```

```
        {
         ch = _inp(PORT1);
         printf("%c",ch);
        }
       if (kbhit())
        {
         ch = _getch();
         _outp(PORT1, ch);
        }
   } while (ch !=27);
}
```

Вначале программный код приложения выполняет настройку параметров UART. Поскольку мы не работаем с прерываниями, то запрещаем их вызов посредством оператора

```
_outp(BR, 0);
```

Далее настроим скорость обмена данными, равную 9600 бод. Чтобы это сделать, необходимо получить доступ к старшему и младшему байту регистра делителя, для чего бит 7 (DLAB) регистра управления линией (`LCR`) следует установить в 1:

```
_outp(LCR, 0x80);
```

Далее запишем в младший байт делителя (`PORT1`) значение `0x0C`, а в старший — 0, что соответствует скорости 9600 бод:

```
_outp(PORT1, 0x0C);
_outp(BR, 0x00);
```

Теперь установим формат данных для передачи: 8 бит, 1 стоповый, без контроля четности. Для этого сбросим бит DLAB и установим младшие два бита регистра управления линией:

```
_outp(LCR, 0x03);
```

Следующие два оператора запрещают работу модема и использование буфера FIFO:

```
_outp(FIFO, 0x0);
_outp(MCR, 0x0);
```

Далее программа переходит к циклическому приему-передаче байтов через устройство UART до тех пор, пока в последовательности данных не встре-

тится символ ESC (0x1B или 27 в десятичной нотации). Наличие байта в регистре приемника проверяется по состоянию бита 0 регистра состояния линии (`LSR`). При установке этого бита в 1 байт считывается в переменную `ch` и выводится на экран консоли функцией `printf`.

Далее программа проверяет, не была ли нажата клавиша (функция `kbhit()`), и если да, то считывает байт данных со стандартного ввода и передает его в регистр передачи устройства UART. Напомню, что для проверки асинхронного приемопередатчика последовательного порта таким способом линии TxD и RxD должны быть соединены.

3.2.2. Настройка и тестирование UART в операционных системах Windows 2000/XP/Vista

При программировании асинхронного приемопередатчика в защищенных операционных системах Windows 2000/XP/2003/Vista следует учитывать, что программа пользователя не имеет прямого доступа к регистрам ввода-вывода физических устройств компьютера, что осложняет задачу непосредственного управления микросхемой UART через регистры устройства.

В таких случаях, если все же необходимо выполнить диагностику и настройку устройства UART, то можно выбрать один из двух вариантов: либо написать собственный драйвер устройства, чтобы можно было манипулировать регистрами асинхронного приемопередатчика, либо воспользоваться специальным программным обеспечением для доступа к регистрам ввода-вывода. Прежде чем детально ознакомиться с методикой программирования устройства UART, нужно четко представить себе, как последовательный порт идентифицируется в операционных системах Windows 2000/XP/2003/Vista.

"С точки зрения" операционной системы UART представляет собой устройство на шине ISA/EISA, которым управляет PnP-контроллер, подключенный к шине PCI или PCI Express. Эту конфигурацию несложно просмотреть, если на пиктограмме **Мой компьютер** выбрать опцию **Управление** и в окне системной консоли в меню **Вид** установить отметку **Устройства по подключению**. Тогда правая панель окна системной консоли будет выглядеть так, как показано на рис. 3.8.

Как видно из примера конфигурации устройств конкретной системы на рис. 3.8, аппаратно последовательный порт (COM1) управляется контроллером I82801, который, в свою очередь, подключен к шине PCI. Таким образом, программный доступ к устройству асинхронного приемопередатчика возмо-

жен через этот контроллер посредством стека драйверов, что исключает доступ приложения пользования непосредственно к регистрам UART.

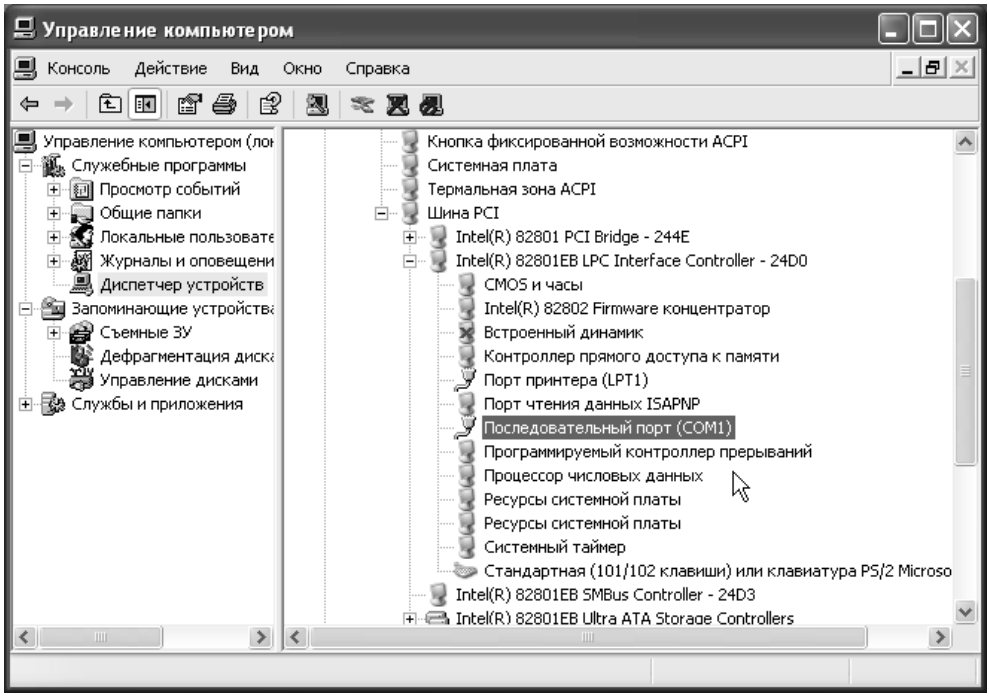


Рис. 3.8. Последовательный порт
в иерархии устройств Windows XP

По умолчанию базовым регистром UART является регистр с адресом 0x3F8 (это соответствует базовым настройкам *BIOS* — Basic Input/Output System) (рис. 3.9).

Таким образом, регистр управления линией `LCR` в такой конфигурации будет иметь адрес 0x3FB, регистр `LSR` — адрес 0x3FD и т. д.

Доступ к регистрам UART имеет только драйвер устройства, который изолирует приложение пользователя от аппаратных ресурсов компьютера. При попытке обращения пользовательского приложения в Windows XP к регистрам какого-либо устройства будет генерироваться общая ошибка защиты, и приложение завершится аварийно. Тем не менее, проблему прямой настройки UART можно решить несколькими относительно простыми способами.

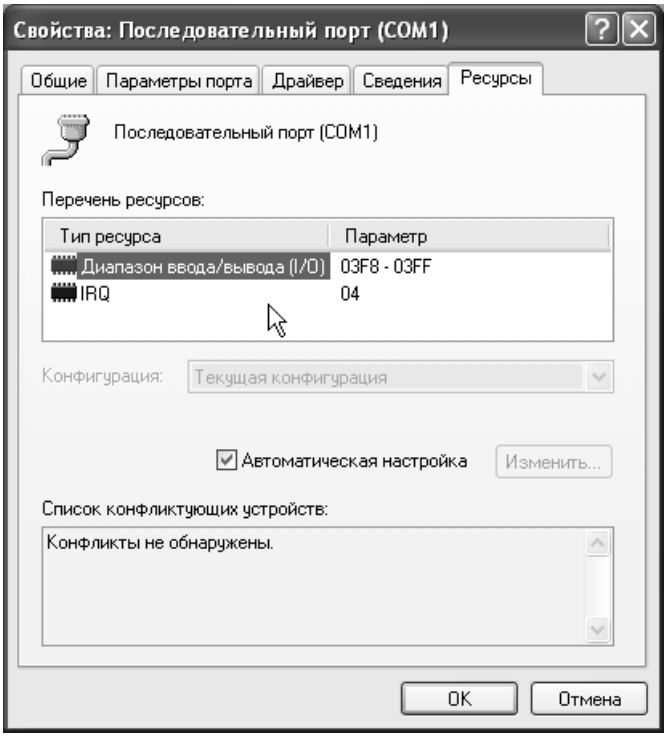


Рис. 3.9. Базовые настройки порта COM1

Первый способ — разработать пользовательское приложение, которое могло бы определенным образом получить доступ к аппаратным ресурсам компьютера, работающего под управлением одной из защищенных операционных систем Windows. Для этих целей ведущими производителями программного обеспечения разработан специальный программный инструментарий, позволяющий приложению пользователя обращаться к регистрам и памяти напрямую.

Хочу обратить внимание на наиболее популярное инструментальное средство *WinDriver* фирмы Jungo (в момент написания книги на рынке присутствовала версия 9.01 этого продукта). Работать с этой программой несложно для программиста средней руки, а 30-дневной демонстрационной версии, которую можно закачать с сайта фирмы, вполне достаточно, чтобы настроить и проверить устройство UART.

Второй способ — разработать простейший драйвер ядра, посредством которого можно было бы получить прямой доступ к регистрам устройства UART.

Мы рассмотрим оба варианта прямого программирования асинхронного приемопередатчика и начнем с WinDriver. Преимуществом работы с WinDriver является то, что здесь не требуется владение техникой разработки драйвера устройства, достаточно знать основы программирования на языке C++ и правильно использовать несколько функций прикладного интерфейса программирования, предоставляемого программой WnDriver.

Программирование UART с помощью программы WinDriver

В основу функционирования программы WinDriver положен принцип, который в упрощенном виде можно описать так: программа WinDriver устанавливает в систему универсальный драйвер ядра (файл windrvr6.sys), который будет обрабатывать вызовы пользовательской программы к физическим устройствам на уровне ядра. Для удобства обращения к драйверу разработаны две библиотеки (высокого и низкого уровня), которые инкапсулируют вызовы пользовательского приложения к физическому устройству, формируя стандартные пакеты запросов для драйвера ядра windrvr6.sys.

Такая реализация интерфейса пользовательской программы и аппаратных ресурсов системы показана на рис. 3.10.

Для выполнения обращения к устройству пользовательское приложение может использовать библиотечные функции WinDriver API, которые начинаются с префикса `WD`. Рассмотрим на практике, как с помощью приложения WinDriver или, вернее, установленных этим приложением компонентов можно управлять функционированием асинхронного приемопередатчика последовательного порта. Здесь можно использовать два пути. Во-первых, применить мастер создания пользовательского приложения Driver Wizard — он сгенерирует для разработчика шаблон приложения, которое затем можно модифицировать для своих потребностей. Во-вторых, создать пользовательское приложение вручную. Второй способ на первый взгляд кажется более трудоемким, однако это не так, и мы сейчас в этом убедимся.

Разработаем приложение пользователя, с помощью которого в Windows XP можно передать в последовательный порт текстовую строку, записывая данные в регистр передачи (он же базовый регистр UART). Разработку приложения выполним в среде Microsoft Visual Studio 2005 (можно использовать и другие версии этой среды разработки) с помощью мастера приложений, выбрав в качестве типа проекта консольное приложение Win32. После создания пустого каркаса приложения включим в него CPP-файл с исходным текстом, приведенным в листинге 3.7.

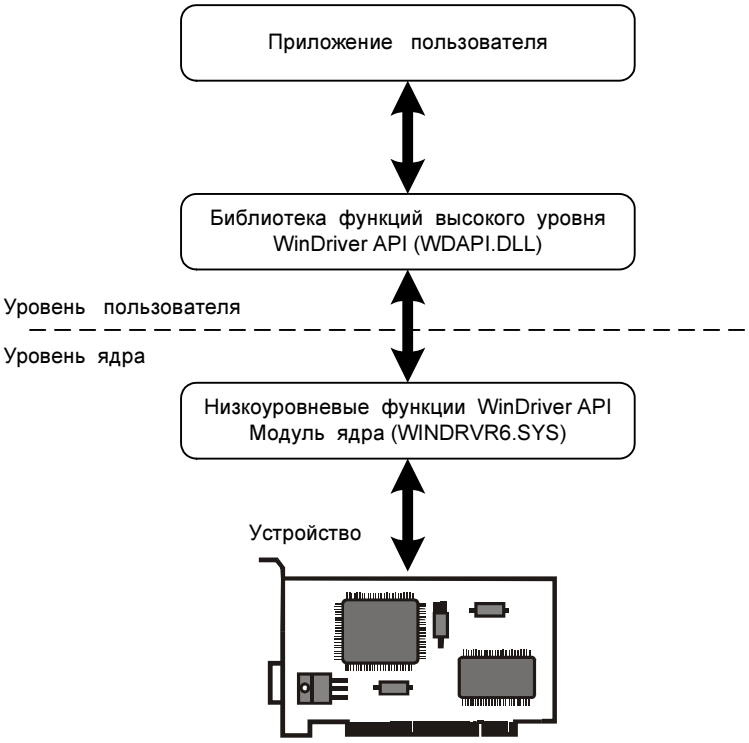


Рис. 3.10. Схема взаимодействия программы пользователя с устройством посредством WinDriver

Листинг 3.7. Передача строки в последовательный порт с использованием регистра UART

```
#include <stdio.h>
#include "windrvr.h"
#include "status_strings.h"

#define PORT1 0x3F8
#define LSR   PORT1 + 5

enum {
    MY_IO_BASE = 0x3F8,
```

```
    MY_IO_SIZE = 0x7
};

static HANDLE hWD;
static WD_CARD_REGISTER cardReg;

static BYTE IO_inp(DWORD dwIOAddr)
{
    WD_TRANSFER trns;
    BZERO(trns);
    trns.cmdTrans = RP_BYTE;
    trns.dwPort = dwIOAddr;
    WD_Transfer( hWD, &trns);
    return trns.Data.Byte;
}

static void IO_outp(DWORD dwIOAddr, BYTE bData)
{
    WD_TRANSFER trns;
    BZERO(trns);
    trns.cmdTrans = WP_BYTE;
    trns.dwPort = dwIOAddr;
    trns.Data.Byte = bData;
    WD_Transfer( hWD, &trns);
}

static BOOL IO_init(void)
{
    WD_VERSION verBuf;
    DWORD dwStatus;

    hWD = WD_Open();
    if (hWD==INVALID_HANDLE_VALUE)
    {
        printf("error opening WINDRVR\n");
```

```
        return FALSE;
    }

    BZERO(verBuf);
    WD_Version(hWD, &verBuf);
    printf(WD_PROD_NAME " version - %s\n", verBuf.cVer);
    if (verBuf.dwVer<WD_VER)
    {
        printf("error incorrect WINDRVR version. needs ver %d\n",WD_VER);
        WD_Close(hWD);
        return FALSE;
    }

    BZERO(cardReg);
    cardReg.Card.dwItems = 1;
    cardReg.Card.Item[0].item = ITEM_IO;
    cardReg.Card.Item[0].fNotSharable = TRUE;
    cardReg.Card.Item[0].I.IO.dwAddr = MY_IO_BASE;
    cardReg.Card.Item[0].I.IO.dwBytes = MY_IO_SIZE;
    cardReg.fCheckLockOnly = FALSE;
    dwStatus = WD_CardRegister(hWD, &cardReg);
    if (cardReg.hCard==0)
    {
        printf("Failed locking device. Status 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        WD_Close(hWD);
        return FALSE;
    }
    return TRUE;
}

static void IO_end(void)
{
    WD_CardUnregister(hWD,&cardReg);
    WD_Close(hWD);
}
```

```
int main(void)
{
    char *str = " String is received by the Serial Terminal";
    char c1;

    if (!IO_init())
        return -1;
    while (*str != 0)
     {
      IO_outp(PORT1, *str++);
      do {
         c1 = IO_inp(LSR) & 0x40;
         } while (c1 == 0);
     }
    IO_end();
    return 0;
}
```

Прежде чем анализировать исходный текст программы, посмотрим, какие изменения нужно внести в настройки нашего проекта, чтобы исходный текст можно было успешно откомпилировать.

Прежде всего, в исходном тексте нужно объявить файлы заголовков windrvr.h и status_strings.h:

```
#include "windrvr.h"
#include "status_strings.h"
```

Затем нужно зайти в окно свойств проекта и внести некоторые изменения. Прежде всего, нужно указать компилятору C++ путь для поиска указанных ранее файлов заголовков (рис. 3.11).

В данном конкретном случае оба файла заголовков находятся в каталоге include установленного на диске I приложения WinDriver.

Затем нужно указать компоновщику (линкеру) имя библиотеки wdapi901.lib (мы используем программу WinDriver версии 9.01), которую следует включить в окончательную сборку (рис. 3.12).

Здесь, как видно из рисунка, выбирается опция **Input** линкера и свойство **Additional dependencies**.

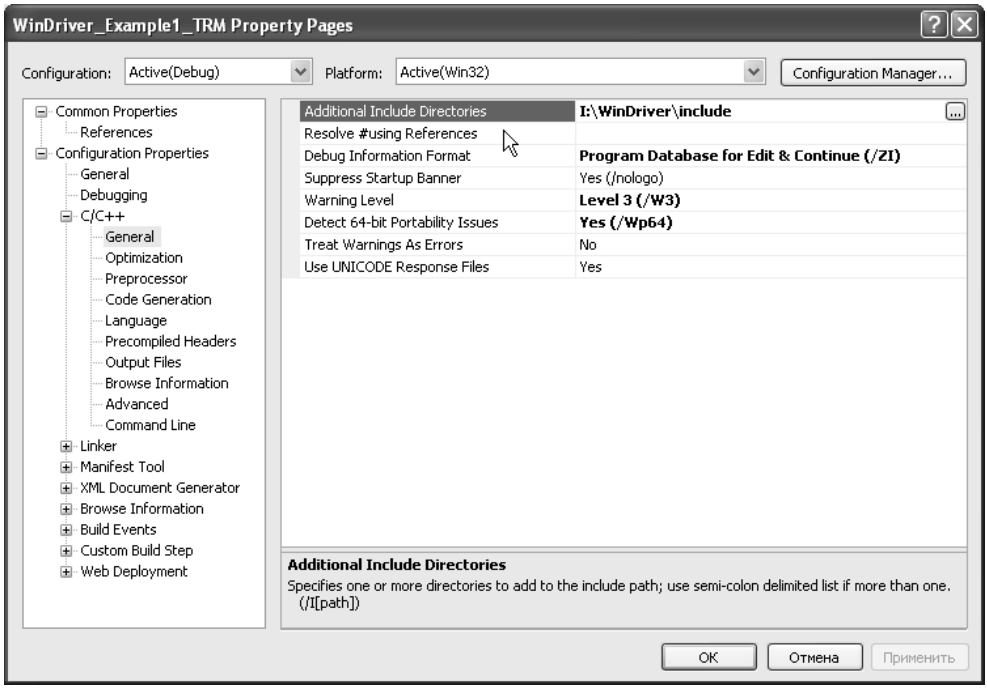


Рис. 3.11. Установка дополнительных каталогов поиска для компилятора C++

Далее, нужно указать компоновщику каталог поиска нужных библиотек (рис. 3.13).

Теперь перейдем к анализу исходного текста программы.

Поскольку наш UART относится к категории ISA-устройств, то программа получается простой. Для приложений с WinDriver обязательно нужно открыть дескриптор объекта драйвера, а в нашем случае нужен еще и дескриптор устройства, поэтому в начале программы объявляются переменные `hWD` и `cardReg`:

```
static HANDLE hWD;
static WD_CARD_REGISTER cardReg;
```

Дескриптор `hWD` будет использоваться при всех обращениях к устройству асинхронного приемопередатчика, а посредством дескриптора `cardReg` можно будет устанавливать параметры ISA-устройства, такие, например, как базовый адрес регистра и число программно-доступных регистров (они определяются константами `MY_IO_BASE` и `MY_IO_SIZE`).

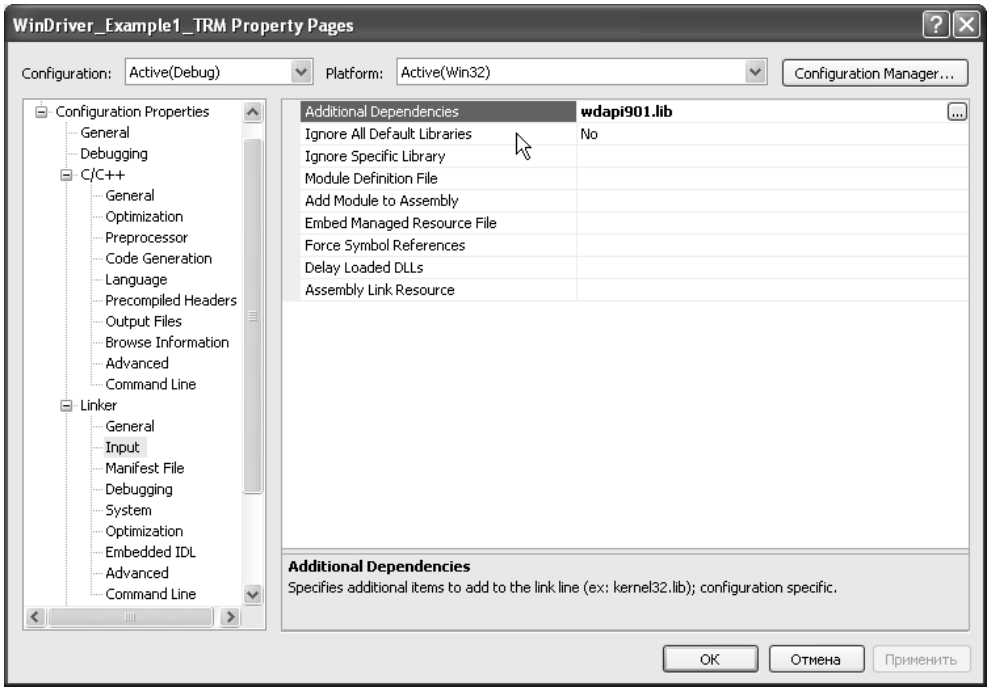


Рис. 3.12. Включение библиотеки wdapi901.lib в проект

В программе определены несколько функций. Функция инициализации `IO_init` выполняет следующие действия:

- открывает дескриптор объекта драйвера (оператор `hWD = WD_Open();`) и, в случае успеха, выводит на консоль версию программы WinDriver. Если не удалось открыть дескриптор объекта, возвращается ошибка, и программа заканчивает работу;
- инициализирует поля структуры `Card`, связанной с дескриптором `cardReg` (устанавливаются базовый адрес регистра устройства и количество доступных регистров, определяется режим монопольного или совместного доступа к устройству);
- выполняет привязку устройства к драйверу windrvr (поскольку это не PnP-устройство (plug and play), то делается вручную). Привязка выполняется в операторе

```
dwStatus = WD_CardRegister(hWD, &cardReg);
```

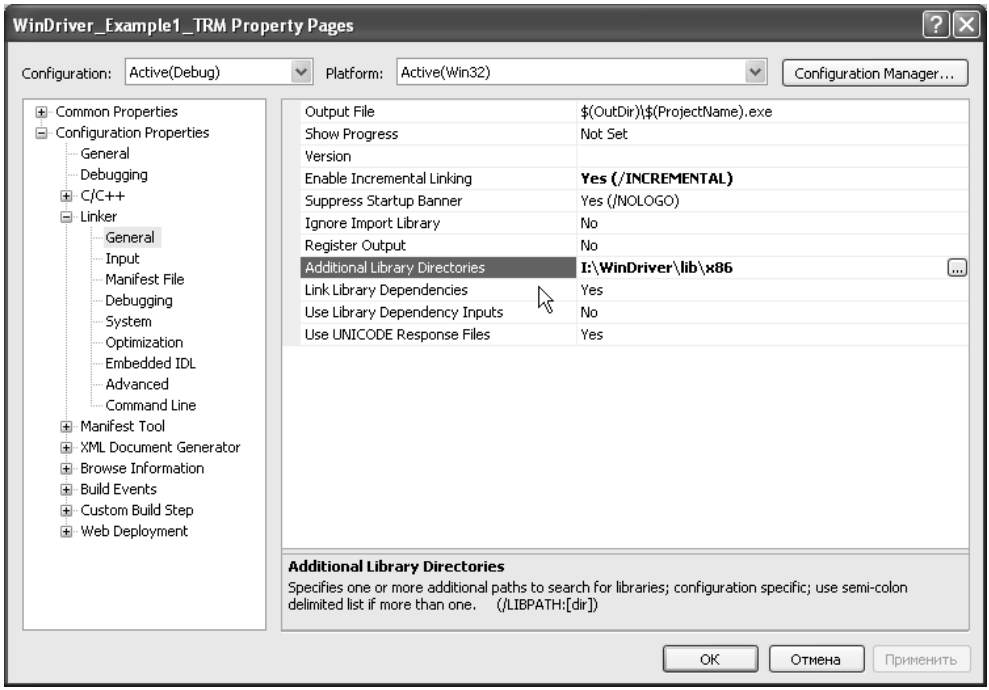


Рис. 3.13. Установка каталога поиска библиотек

В случае успешного завершения функции `WD_CardRegister` можно выполнять операции обмена данными с устройством, иначе дескриптор `hWD` закрывается и программа завершается.

По завершению работы с устройством следует разблокировать его и закрыть дескриптор драйвера, что выполняют операторы функции `IO_end`:

```
WD_CardUnregister(hWD,&cardReg);
WD_Close(hWD);
```

В программе объявлены еще две функции — `IO_inp` и `IO_outp`, с помощью которых выполняется чтение данных в устройство и вывод данных из устройства соответственно. В обеих функциях объявлена переменная `trns` типа `WD_TRANSFER`, представляющая собой структуру, поля которой определяют направление передачи данных (`cmdTrans`) и адрес порта (регистра), который используется для обмена данными (`dwPort`).

Например, в функции `IO_inp`, выполняющей чтение данных, этим полям присваиваются следующие значения:

```
trns.cmdTrans = RP_BYTE;
trns.dwPort = dwIOAddr;
```

Здесь значение `RP_BYTE` указывает на то, что будет выполняться чтение байта данных, а `dwIOAddr` определяет адрес регистра (порта), с которого будет выполняться чтение.

В функции `IO_outp` значение `WP_BYTE` поля `cmdTrans` указывает на то, что будет выполняться запись байта в регистр с адресом `dwIOAddr`.

Поле `Data.Byte` структуры `trns` при выводе (записи) должно содержать байт данных (функция `IO_outp`), а при чтении в него помещается результат (функция `IO_inp`).

Передача данных в обоих направлениях осуществляется библиотечной функцией `WD_Transfer` в операторе

```
WD_Transfer( hWD, &trns);
```

Основная программа выполняет три следующие действия:

1. С помощью функции `IO_Init` выполняется инициализация необходимых объектов и структур.
2. Выполняется передача строки байтов в последовательный порт в цикле `while`.
3. С помощью функции `IO_end` уничтожаются дескрипторы объектов, и происходит выход из программы.

Остановимся более подробно на алгоритме передачи строки байтов. Алгоритм реализован в цикле `while`:

```
while (*str != 0)
     {
      IO_outp(PORT1, *str++);
      do {
         c1 = IO_inp(LSR) & 0x40;
         } while (c1 == 0);
     }
```

Поскольку строка `str` является стандартной строкой с завершающим нулем, то выход из цикла `while` происходит по достижению нулевого байта строки. Запись (передача) одного байта происходит с помощью функции `IO_outp`, в которой передаваемый байт помещается в регистр приема/передачи устрой-

ства UART, являющийся базовым для данного устройства и определенный константой PORT1. В качестве второго параметра функции IO_outp служит адрес (указатель) очередного байта строки, который тут же инкрементируется для доступа к очередному байту данных в следующей итерации.

Напомню, что байт, помещенный в регистр передачи UART, передается в линию посредством регистра сдвига, что требует определенного времени. О полном завершении передачи байта (регистры передачи и сдвига будут пустыми) свидетельствует флаг завершения передачи (бит 6 регистра состояния линии LSR устройства UART), который будет установлен в 1. В цикле do осуществляется проверка этого флага, и если он установлен (переменная c1 равна 1), то начинается передача следующего байта и т. д.

Это все, что касается работы этой программы. Последнее замечание: при разработке приложения следует проверить базовый адрес используемого порта — он может отличаться от значения 0x3F8.

Тестирование работоспособности UART последовательного порта можно выполнить, соединив выводы TxD и RxD. После этого запускаем любую терминальную программу, работающую с COM-портом, и настраиваем ее для работы с COM1. Затем запускаем откомпилированное приложение. Если все сделано правильно, то в окне терминальной программы появится содержимое строки str.

По схожей схеме можно реализовать и проверку приема данных устройством UART, но исходный текст приложения будет другим (листинг 3.8).

Листинг 3.8. Прием данных в последовательный порт с использованием регистра UART

```
#include <stdio.h>
#include "windrvr.h"
#include "status_strings.h"

#define PORT1 0x3F8
#define LSR   PORT1 + 5

enum {
    MY_IO_BASE = 0x3F8,
    MY_IO_SIZE = 0x7
};
```

```
static HANDLE hWD;
static WD_CARD_REGISTER cardReg;

static BYTE IO_inp(DWORD dwIOAddr)
{
    WD_TRANSFER trns;
    BZERO(trns);
    trns.cmdTrans = RP_BYTE;
    trns.dwPort = dwIOAddr;
    WD_Transfer( hWD, &trns);
    return trns.Data.Byte;
}

static void IO_outp(DWORD dwIOAddr, BYTE bData)
{
    WD_TRANSFER trns;
    BZERO(trns);
    trns.cmdTrans = WP_BYTE;
    trns.dwPort = dwIOAddr;
    trns.Data.Byte = bData;
    WD_Transfer( hWD, &trns);
}

static BOOL IO_init(void)
{
    WD_VERSION verBuf;
    DWORD dwStatus;

    hWD = WD_Open();
    if (hWD==INVALID_HANDLE_VALUE)
    {
        printf("error opening WINDRVR\n");
        return FALSE;
    }

    BZERO(verBuf);
    WD_Version(hWD, &verBuf);
```

```
    printf(WD_PROD_NAME " version - %s\n", verBuf.cVer);
    if (verBuf.dwVer<WD_VER)
    {
        printf("error incorrect WINDRVR version. needs ver %d\n",WD_VER);
        WD_Close(hWD);
        return FALSE;
    }

    BZERO(cardReg);
    cardReg.Card.dwItems = 1;
    cardReg.Card.Item[0].item = ITEM_IO;
    cardReg.Card.Item[0].fNotSharable = TRUE;
    cardReg.Card.Item[0].I.IO.dwAddr = MY_IO_BASE;
    cardReg.Card.Item[0].I.IO.dwBytes = MY_IO_SIZE;
    cardReg.fCheckLockOnly = FALSE;
    dwStatus = WD_CardRegister(hWD, &cardReg);
    if (cardReg.hCard==0)
    {
        printf("Failed locking device. Status 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        WD_Close(hWD);
        return FALSE;
    }

    return TRUE;
}

static void IO_end(void)
{
    WD_CardUnregister(hWD,&cardReg);
    WD_Close(hWD);
}

int main(void)
{
   char buf[256];
```

```
   char *pbuf = buf;
   char c1;

   if (!IO_init())
        return -1;
    printf("Waiting for data...\n");
    while (1)
     {
      do {
          c1 = IO_inp(LSR) & 0x1;
         } while (c1 == 0);
     *pbuf = IO_inp(PORT1);
      printf("%c", *pbuf++);
     }

   IO_end();
   return 0;
}
```

Назначение структур, переменных и функций, а также дополнительные настройки компилятора и компоновщика мы детально обсудили в предыдущем примере, поэтому остановимся на реализации алгоритма приема данных устройством UART. Прием байта данных асинхронным приемопередатчиком последовательного порта осуществляется в цикле `while(1)` основной программы.

Поступающие через порт COM1 байты данных сохраняются в буфере `buf` и адресуются указателем `pbuf`. Признаком того, что данные приняты и находятся в регистре приема/передачи, является установленный бит 0 регистра состояния линии (`LSR`) микросхемы UART. Состояние этого бита проверяется в цикле `do`:

```
do {
    c1 = IO_inp(LSR) & 0x1;
   } while (c1 == 0);
```

При установке бита 0 регистра `LSR` в 1 байт данных можно считать в буфер `buf` и вывести на консоль, что и выполняют операторы

```
*pbuf = IO_inp(PORT1);
 printf("%c", *pbuf++);
```

Как и в предыдущем примере, можно запустить терминальную программу и наше приложение. Набираемые в окне терминальной программы символы будут отображаться в окне нашего приложения. Естественно, что для выполнения тестирования выводы TxD и RxD последовательного порта должны быть соединены.

Если вас не устраивают возможности программы WinDriver и вы хотели бы работать с физическим устройством напрямую, то можете разработать драйвер ядра Windows. Естественно, что такая разработка требует от программиста достаточно высокой квалификации, поскольку техника программирования драйверов существенно отличается от разработки обычных приложений. К счастью, фирма Microsoft предоставляет разработчику инструментальное средство разработки драйверов ядра — *Driver Development Kit* (DDK), которым мы воспользуемся для создания простейшего драйвера ядра. Он позволит программировать и тестировать устройство UART напрямую. В следующем разделе мы увидим, как это сделать.

Настройка и тестирование UART посредством драйвера устройства

В этом разделе мы вкратце ознакомимся с техникой разработки драйверов устройств и создадим простейший драйвер для управления нашим устройством UART. Теория разработки драйверов довольно сложная и занимает многие сотни страниц документации, разработанной Microsoft. Мы рассмотрим только основные моменты процесса проектирования драйверов устройств, которых будет вполне достаточно для создания простейших драйверов для доступа к устройствам ПК, в частности, к устройству UART.

Разработка драйверов ядра для операционных систем Windows считается чем-то из разряда "черной магии". Большинство программистов и пользователей полагают, что это чрезвычайно трудное занятие и доступно далеко не каждому. В определенном смысле это действительно так, если, например, пытаться разработать драйвер звуковой или видеокарты. Для управления относительно простыми устройствами, которые требуют чтения-записи нескольких регистров, как, например, устройство UART или параллельный порт ПК, разработать несложный драйвер может даже пользователь средней руки, достаточно хорошо владеющий основами программирования на языке C и понимающий принципы организации защищенных операционных систем Windows 2000/XP/2003/Vista.

Следует помнить, что драйверы устройств работают в режиме ядра и могут получить доступ ко всем аппаратно-программным ресурсам системы без каких-либо ограничений.

> ***ВНИМАНИЕ!***
>
> Неправильно спроектированный драйвер может привести к мгновенному краху системы ("синий экран смерти"). Если вы захотите модифицировать приведенный в этой главе программный код драйверов или написать собственный, то предварительно изучите документацию по разработке драйверов, чтобы избежать неприятных последствий.

Для разработки драйверов будем применять стандартный свободно распространяемый программный пакет Microsoft Windows DDK (Driver Development Kit). Кроме компилятора и вспомогательных утилит, в состав DDK включена обширная документация вместе с примерами разработок драйверов.

Взаимодействие пользовательского приложения и устройства в операционной системе в упрощенном виде показано на рис. 3.14.

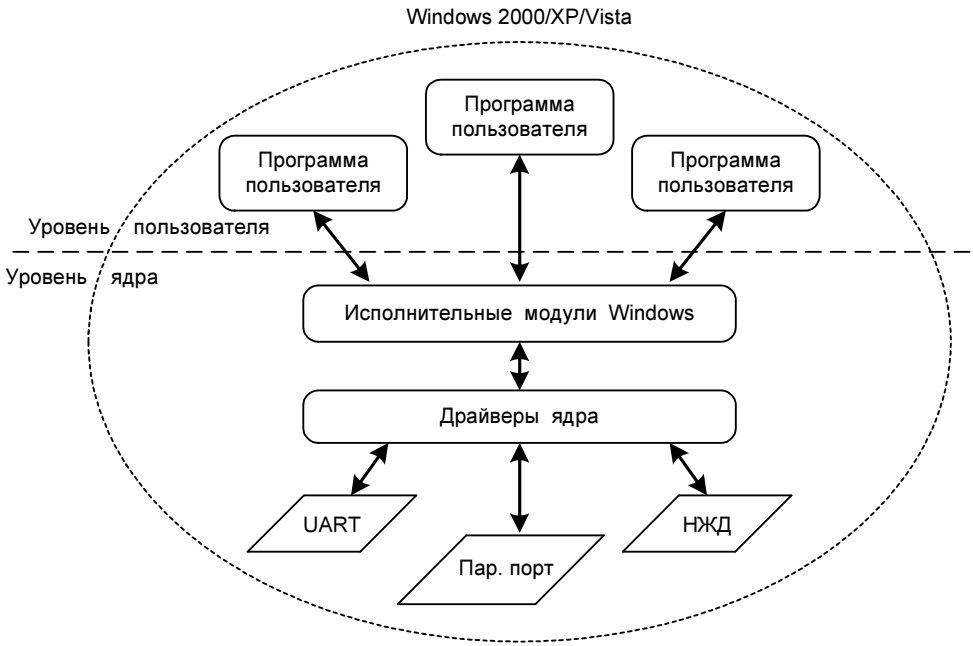


Рис. 3.14. Взаимодействие программ пользователя с устройствами в операционной системе Windows

В защищенных операционных системах Windows 2000/XP/2003/Vista программы пользователя взаимодействуют с любым физическим устройством (последовательным принтером, сканером и т. д.) исключительно через драйверы устройств. Программа пользователя, которой требуется доступ к устройству, обращается к исполнительным модулям Windows, представляющим собой программный интерфейс между пользователем и системой. Исполнительный модуль формирует при необходимости запрос к другим модулям системы или к драйверу устройства, к которому выполняется обращение. Подобная иерархическая структура взаимодействия программы пользователя и устройства призвана обеспечить устойчивое функционирование операционной системы, которое не может быть нарушено какими-то операциями со стороны пользовательского приложения.

Такая иерархия в операционных системах Windows реализуется посредством присвоения привилегий любой работающей программе, или по-другому, *процессу*. Уровень привилегий определяет, может ли работающая программа (процесс) обращаться напрямую к системным ресурсам, таким, как физические адреса памяти, порты ввода-вывода, линии прерывания, устройства прямого доступа к памяти и т. д. Для программ пользователя, какими являются обычные приложения, доступ к ресурсам системы запрещен. Если из приложения пользователя выполнить обращение к аппаратным регистрам, то будет сгенерирована ошибка защиты.

Пример исходного текста простейшего консольного приложения, написанного в Visual C++, представлен в листинге 3.9.

Листинг 3.9. Пример доступа к параллельному порту из приложения пользователя

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
  _outp(0x378, 0x0);
}
```

Если попробовать запустить эту программу, в которой выполняется запись байта данных в регистр параллельного порта, имеющего адрес 0x378, на компьютере под управлением Windows XP, то будет сгенерирована ошибка со следующим сообщением (рис. 3.15).

Рис. 3.15. Ошибка программы при попытке записи в регистр параллельного порта

Команды ассемблера `in` и `out` являются "привилегированными" инструкциями процессора Intel и могут выполняться только программами, работающими на уровне ядра системы, поэтому пользовательская программа и завершается аварийным образом. К ресурсам системы напрямую могут обращаться некоторые системные программы и большинство драйверов устройств.

Таким образом, для обращения к аппаратным ресурсам устройства из пользовательского приложения можно использовать программные средства наподобие WinDriver, применение которого мы анализировали ранее, или же можно написать драйвер устройства.

Вторую возможность мы сейчас и попробуем реализовать на практике. В нашем конкретном случае необходимо написать драйвер устройства, который мог бы напрямую обращаться к регистрам асинхронного приемопередатчика UART, и создать приложение, которое работало бы с этим драйвером устройства для выполнения операций ввода-вывода.

Прежде чем реализовать подобный проект, нужно хотя бы вкратце изучить некоторые ключевые аспекты взаимодействия программы пользователя и драйвера устройства, а также базовые принципы разработки драйверов устройств.

Начнем с того, что в операционных системах Windows все физические устройства условно разделяются на две большие группы: поддерживающие технологию PnP (plug and play) или не поддерживающие ее. Соответственно, драйверы, поддерживающие технологию PnP, называют *WDM-драйверами*, а драйверы, не поддерживающие эту технологию — *NT-драйверами*. Это весьма упрощенный подход, но при первоначальном изучении данной темы этого достаточно. Оба типа драйверов в своей основе используют одни и те же

функции, но WDM-драйвер включает ряд дополнительных функций, позволяющих реализовать возможности технологии PnP (установка-удаление без перезагрузки, управление энергопотреблением и т. д.). Для установки/обновления WDM-драйвера требуется наличие INF-файла, в котором описаны характеристики драйвера (имя файла, место установки, данные о производителе и т. д.).

Для установки NT-драйвера INF-файл не нужен, но требуется написать программу, позволяющую с помощью специальных функций WIN API включить драйвер в систему. То же самое, в принципе, можно сделать и вручную, прописав соответствующие значения ключей в системном реестре. Этот тип драйвера используется для работы с устройствами, не поддерживающими технологию PnP, а также при написании драйверов, не работающих с каким-либо оборудованием. В частности, NT-подобный драйвер можно использовать при работе с параллельным, последовательным портом, звуковой картой и другими устройствами в системе. Такой драйвер сравнительно несложно написать, что мы и продемонстрируем в последующих примерах.

NT-совместимый драйвер устройства лучше всего представить в виде программного "контейнера", содержащего отдельные программные модули (рис. 3.16).

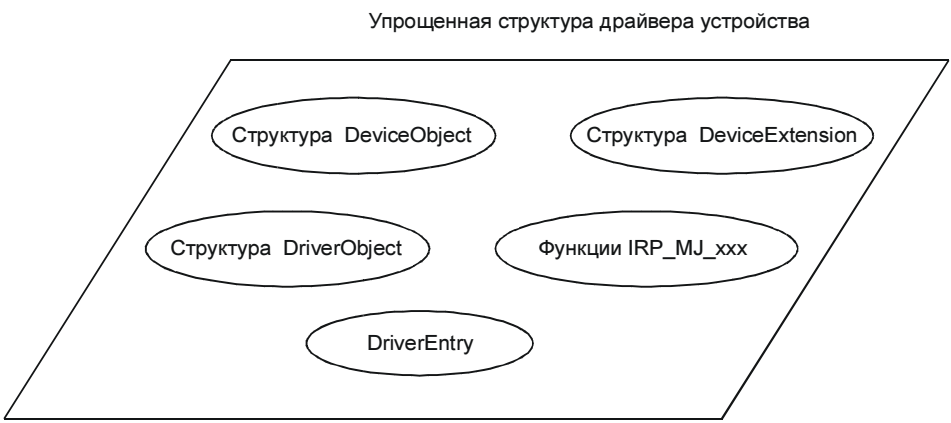


Рис. 3.16. Упрощенная программная модель драйвера устройства

Если прибегнуть к аналогии, то драйвер устройства по структуре напоминает библиотеку динамической компоновки (DLL) операционной системы Windows. Как и DLL, он содержит набор функций вида IRP_MJ_xxx, каждая из которых выполняет ту или иную операцию с устройством. В драйвере нет точки входа в программу, как например, `main` в программе на языке C. Для

инициализации драйвера и связанных с ним программных структур служит функция `DriverEntry`, которая вызывается операционной системой в момент загрузки драйвера.

Это весьма упрощенное описание структуры драйвера устройства, но оно дает представление о том, как он работает. Жизненный цикл функционирования драйвера устройства можно разделить (условно) на три этапа:

1. Инициализация.
2. Выполнение операций по запросам операционной системы на ввод-вывод данных.
3. Остановка и удаление из системы.

Драйвер устройства записан на диске в формате файла с расширением sys. Для операционных систем Windows общепринято располагать файлы драйверов в каталоге windows\system32\drivers, хотя это и не обязательно. Сведения об установленных драйверах хранятся в системном реестре Windows в ключе \registry\machine\system\CurrentControlcet\Services. Исторически сложилось так, что записи о системных службах и драйверах хранятся в одном и том же разделе реестра (services).

Помимо всего прочего, записи содержат тип запуска драйвера — по нему система определяет, когда нужно загружать драйвер. Здесь есть очень важный момент: при испытаниях работоспособности драйвера желательно не инициализировать драйвер в момент загрузки системы, иначе можно вообще не загрузиться, если в драйвере имеется какая-либо ошибка.

При инициализации драйвера операционная система загружает двоичный образ дискового файла в защищенную область памяти и передает управление функции `DriverEntry`. Основная задача этой функции — проинициализировать поля структур DriverObject и DeviceObject и выполнить привязку устройства к пространству имен операционной системы, чтобы можно было обращаться каким-то образом из программы пользователя к устройству. Кроме этого, здесь же инициализируется массив указателей на функции вида IRP_MJ_xxx, где xxx обозначает операцию, выполняемую данной функцией.

Например, указатель IRP_MJ_CREATE может ссылаться на функцию, которая будет вызываться при создании (открытии) устройства (например, `CreateFile()` из программы пользователя). Каждый драйвер может включать различные функции, в зависимости от назначения устройства. Например, в устройство можно только записывать данные, но нельзя читать — в этом случае можно не определять функцию для IRP_MJ_READ или сделать ее в виде программной "заглушки".

Как в драйвере выбираются функции, которые должны выполнять в данный момент операцию с устройством? Для этого предусмотрен механизм пакетов на запрос ввода-вывода (I/O Request Packet). Например, если программа пользователя выполняет запись данных в устройство при помощи WINAPI-функции `WriteFile()`, то операционная система формирует пакет запроса IRP_MJ_WRITE и посылает его драйверу устройства для выполнения.

В свою очередь, драйвер устройства вызывает функцию, чей адрес указан при инициализации массива указателей, и передает ей управление. Любой драйвер, тем не менее, должен включать обработчики пакетов запроса IRP_MJ_CREATE (открытие устройства) и IRP_MJ_CLOSE (закрытие устройства).

Пакет запроса IRP содержит необходимые поля и адреса областей памяти, выделенных системой для временного хранения данных при чтении-записи. Мы не будем детально рассматривать структуру IRP, а перейдем к анализу взаимодействия приложения и устройства посредством драйвера.

Приложение пользователя, работающее в Windows, может взаимодействовать с устройством через исполнительный уровень, на котором работают важные системные службы, посредством обращения к функциям интерфейса прикладного программирования (функции WIN API), включенным в некоторые библиотеки динамической компоновки, например, в kernel32.dll. Здесь нужно отметить следующий важный момент: все устройства, с которыми взаимодействует операционная система, представлены в ней как файлы, причем это относится не только к файлам в их традиционном представлении (определенным образом скомпонованные группы данных на жестком диске), но и ко всем аппаратным устройствам. Так, например, параллельный порт ПК в операционной системе Windows может быть представлен как файл с именем "LPT1", последовательному порту может быть присвоено имя "COM1" и т. д.

Рассмотрим, например, как может осуществляться взаимодействие программы пользователя с устройством, подключенным к последовательному порту COM1 (рис. 3.17).

Схема взаимодействия пользовательского приложения и устройства последовательного ввода-вывода, изображенная на рис. 3.17, является весьма показательной и характерна для всех устройств операционной системы.

Чтобы начать работу с устройством (в данном случае, это последовательный порт COM1), пользовательское приложение должно открыть его с помощью функции WIN API `CreateFile()`. Эта функция всегда должна выполняться первой при обращении к объекту файловой системы, будь то дисковый файл

или устройство ввода-вывода. В операционных системах Windows (как и в UNIX) все объекты системы (дисковые файлы, устройства ввода-вывода, именованные каналы и т. д.) интерпретируются как файлы, и операции над ними выполняются с помощью одних и тех же функций прикладного интерфейса программирования WIN API. Функции WIN API транслируются в функции исполнительных модулей, которые, в свою очередь, обращаются к функциям драйвера устройства.

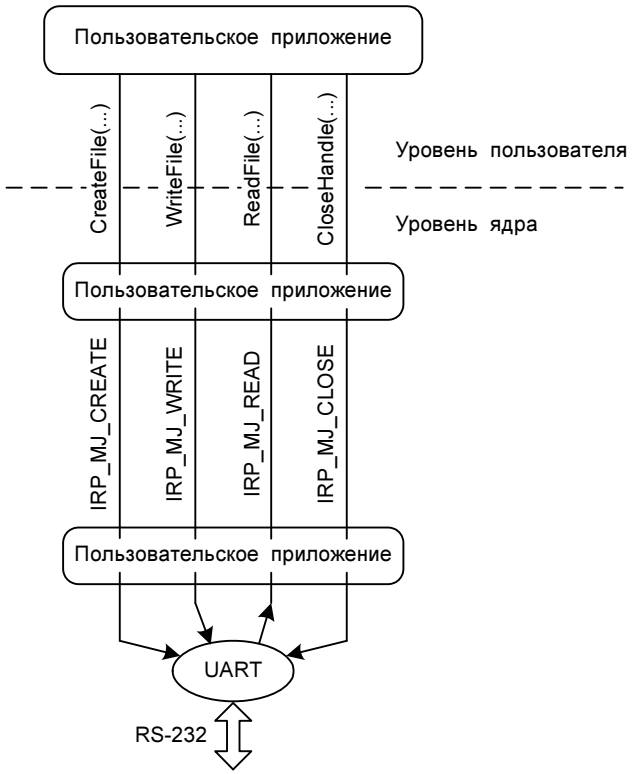


Рис. 3.17. Схема выполнения операций с устройством

Обращения к устройствам в драйверах устройств осуществляются с помощью универсального механизма *пакетов запросов — IRP* (I/O Request Packet). IRP можно представить в виде структуры, в которую включается вся необходимая информация для осуществления операции с устройством. Пакет запроса формируется исполнительным модулем операционной системы, вернее, *Менеджером ввода-вывода* этого модуля, который по типу обращения из

пользовательской программы (открытие, закрытие, чтение или запись) определяет, что должно содержаться в пакете запроса IRP. Менеджер ввода-вывода является важным программным компонентом, осуществляющим взаимодействие программы пользователя и драйвера устройства. Менеджер ввода-вывода — это обобщенное название системных сервисов и программ, осуществляющих формирование стандартных пакетов запросов (IRP) к драйверу устройства, буферизацию данных ввода-вывода устройства и синхронизацию очередей ввода-вывода. Замечу, что процесс выполнения запросов намного более сложен и включает дополнительные шаги, но для ясности изложения я даю последовательность шагов в упрощенном виде.

Рассмотрим, что происходит, например, при попытке записи данных в устройство из программы пользователя с помощью функции WIN API `WriteFile` (предполагаем, что устройство уже открыто с помощью функции `CreateFile` и получен его дескриптор):

1. Менеджер ввода-вывода исполнительного модуля определяет, что нужно передать определенное количество байтов данных в устройство, и начинает формировать пакет запроса IRP, инициализируя поля этой структуры определенными значениями и адресами.
2. Менеджер ввода-вывода вызывает функцию, определенную вектором IRP_MJ_WRITE, которая будет обрабатывать данный запрос. В качестве параметров этой функции передаются указатель на структуру `DeviceObject`, связанную с устройством, и указатель на пакет запроса IRP.
3. Менеджер ввода-вывода ожидает завершения выполнения запроса и возвращает статус операции программе пользователя.

При выполнении операции чтения данных из устройства пользовательское приложение вызывает функцию `ReadFile`. Менеджер ввода-вывода также формирует пакет запроса IRP, но для выполнения этого запроса вызывает функцию с кодом IRP_MJ_READ. После выполнения запроса Менеджер ввода-вывода, кроме статусной информации, передает приложению пользователя данные (если чтение данных из устройства было успешным).

По схожей схеме выполняется открытие устройства функцией `CreateFile` (код функции IRP_MJ_CREATE) и закрытие устройства (код функции IRP_MJ_CLOSE).

Это весьма упрощенный пример взаимодействия программы пользователя и устройства ввода-вывода, но все же он дает определенное представление о механизмах такого взаимодействия.

Вместо функций `ReadFile()` и `WriteFile()` имеется возможность использовать WIN API-функцию `DeviceIoControl()`. С помощью этой функции можно послать устройству код какой-либо операции, не только записи или чтения данных, но также любой другой код, на который устройство может реагировать. Это расширяет возможности операций ввода-вывода, но требует, чтобы драйвер устройства мог обрабатывать такие коды. В наших последующих примерах мы будем использовать именно функцию `DeviceIoControl()`.

На этом теоретическую часть основ разработки NT-подобных драйверов устройств мы закончим. Заинтересованных в дальнейшем изучении техники написания драйверов могу отослать к документации, входящей в состав Windows DDK.

Сейчас мы на практике посмотрим, как можно реализовать простейший драйвер устройства для нашего асинхронного приемопередатчика UART. Все, что нам нужно от драйвера — это то, чтобы можно было записать данные в регистр и прочитать их из регистра.

Наш программный проект будет состоять из драйвера устройства, посредством которого пользовательское приложение будет посылать устройству команды, и собственно программы пользователя. В программе пользователя для обращения к устройству мы будем использовать функцию WIN API `DeviceIoControl`.

Для создания драйвера устройства Windows на компьютере должен быть установлен пакет Windows DDK (Driver Development Kit) фирмы Microsoft, желательно последней версии (на момент написания книги таковой являлась версия 2003 SP1). Этот пакет включает все необходимые средства для разработки и отладки драйверов. Кроме того, нужно иметь под рукой хороший компилятор C/C++ (подойдет бесплатная версия Microsoft Visual Studio Express Edition) для разработки приложения для тестирования драйвера.

Кроме того, сам драйвер устройства должен быть проинсталлирован в операционной системе. Можно установить драйвер, написав небольшую программу установки, но для этого придется изучить принципы работы Менеджера установки системных сервисов (*SCM — Service Control Manager*). Поэтому для установки и удаления драйвера лучше использовать одну из множества утилит, ссылки на которые приводятся в Интернете. Можно применить очень удобный инсталлятор, доступный на сайте **www.beyondlogic.org**.

Исходный текст драйвера нужно сохранить в файле comdriver.c. Сам исходный текст, который мы детально проанализируем, приводится в листинге 3.10.

Листинг 3.10. Исходный текст простого драйвера устройства для работы с UART

```
#include <ntddk.h>
#include <conio.h>

#define IOCTL_WBYTE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                             0x801,\
                             METHOD_BUFFERED,\
                             FILE_ANY_ACCESS)

#define IOCTL_RBYTE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                             0x802,\
                             METHOD_BUFFERED,\
                             FILE_ANY_ACCESS)

#define NT_DEVICE_NAME L"\\Device\\comdevice"
#define DOS_DEVICE_NAME L"\\DosDevices\\comdevice"

typedef struct _MY_DEVICE_EXTENSION
{
    USHORT Addr;
    UCHAR Data;
} MY_DEVICE_EXTENSION, *PMY_DEVICE_EXTENSION;

VOID
comdevice_Unload (IN PDRIVER_OBJECT DriverObject)
{
   UNICODE_STRING DosDeviceName;
   RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
   IoDeleteSymbolicLink(&DosDeviceName);
   IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS
comdevice_Create(IN PDEVICE_OBJECT DeviceObject,
```

```
                IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
comdevice_Close(IN PDEVICE_OBJECT DeviceObject,
               IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
comdevice_IOCTL (IN PDEVICE_OBJECT DeviceObject,
                 IN PIRP Irp)
{
   PMY_DEVICE_EXTENSION dx =
(PMY_DEVICE_EXTENSION)DeviceObject->DeviceExtension;
   PIO_STACK_LOCATION pIoStack = IoGetCurrentIrpStackLocation(Irp);
   ULONG ctlCode = pIoStack->Parameters.DeviceIoControl.IoControlCode;

   PUCHAR pBuf = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;
   NTSTATUS status;

switch (ctlCode)
  {
    case IOCTL_WBYTE:
       RtlMoveMemory(&dx->Addr,
                     pBuf,
                     2);
       RtlMoveMemory(&dx->Data,
                     pBuf+2,
```

```
                    1);
      _outp(dx->Addr, dx->Data);
    case IOCTL_RBYTE:
       RtlMoveMemory(&dx->Addr,
                    pBuf,
                    2);
       dx->Data = _inp(dx->Addr);
       RtlMoveMemory(pBuf,
                    &dx->Data,
                    1);
      Irp->IoStatus.Information = 1;
  }
Irp->IoStatus.Status = STATUS_SUCCESS;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return STATUS_SUCCESS;
}

NTSTATUS
DriverEntry (IN PDRIVER_OBJECT DriverObject,
             IN PUNICODE_STRING RegistryPath)
{
  PDEVICE_OBJECT DeviceObject;
  UNICODE_STRING NtDeviceName;
  UNICODE_STRING DosDeviceName;
  NTSTATUS status;

  RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME);
  status = IoCreateDevice(DriverObject,
                          sizeof(MY_DEVICE_EXTENSION),
                          &NtDeviceName,
                          FILE_DEVICE_UNKNOWN,
                          0,
                          FALSE,
                          &DeviceObject);
  if (!NT_SUCCESS(status))
```

```
    {
     IoDeleteDevice(DeviceObject);
     return status;
    }

  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
  status = IoCreateSymbolicLink(&DosDeviceName, &NtDeviceName);
  if (!NT_SUCCESS(status))
    {
     IoDeleteSymbolicLink(&DosDeviceName);
     IoDeleteDevice(DeviceObject);
     return status;
    }

  DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
  DeviceObject->Flags |= DO_BUFFERED_IO;

  DriverObject->MajorFunction[IRP_MJ_CREATE] = comdevice_Create;
  DriverObject->MajorFunction[IRP_MJ_CLOSE]  = comdevice_Close;
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = comdevice_IOCTL;
  DriverObject->DriverUnload = comdevice_Unload;

  return STATUS_SUCCESS;
}
```

Анализ исходного текста начнем с описания переменных и констант, используемых в драйвере. В начало листинга обязательно включается файл заголовка ntddk.h, в котором определены все необходимые функции. Кроме того, в нашем драйвере мы будем записывать байты данных в регистр и читать их из регистра с помощью функций `_outp` и `_inp`, поэтому в листинг следует включить и файл заголовка conio.h, где эти функции определены.

Драйвер устройства будет работать с пользовательским приложением, в котором применяется функция `DeviceIoControl()`. В этом случае как в пользовательском приложении, так и в драйвере устройства должны быть определены одни и те же команды, используемые в функции `DeviceIoControl()`. В данном примере будут применяться две команды: одна для записи данных и другая для чтения.

Для объявления обеих команд используется макрос CTL_CODE:

```
#define IOCTL_WBYTE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                             0x801,\
                             METHOD_BUFFERED,\
                             FILE_ANY_ACCESS)

#define IOCTL_RBYTE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                             0x802,\
                             METHOD_BUFFERED,\
                             FILE_ANY_ACCESS)
```

Второй параметр команд должен быть числовым и превышать значение 0x800 (пользовательский диапазон). Напомню, что коды IOCTL-команд, а также их атрибуты в приложении и в драйвере должны совпадать. Обратите внимание на то, что в коде IOCTL-команды указан буферизованный метод данных, т. е. при передаче данных от устройства к драйверу и наоборот Менеджер ввода-вывода выделит буфер памяти для временного хранения данных.

Строки

```
#define NT_DEVICE_NAME L"\\Device\\comdevice"
#define DOS_DEVICE_NAME L\\DosDevices\\comdevice
```

в формате UNICODE определяют имя устройства (comdevice), к которому будут выполняться обращения. Строка NT_DEVICE_NAME указывает имя устройства таким, каким оно должно быть в пространстве имен операционной системы. Константная строка DOS_DEVICE_NAME нужна для того, чтобы WIN API-функция CreateFile() в приложении пользователя могла обратиться к данному устройству. Функция CreateFIle() "не видит" устройство, определяемое через константу NT_DEVICE_NAME, но драйвер устройства обращается к устройству как раз по этому имени.

Чтобы разрешить это противоречие, используется функция IoCreateSymbolicLink, которая связывает оба имени.

Для хранения данных в драйвере применяется область расширения (Device Extension), которая определяется следующим образом:

```
typedef struct _MY_DEVICE_EXTENSION
{
    USHORT Addr;
    UCHAR Data;
} MY_DEVICE_EXTENSION, *PMY_DEVICE_EXTENSION;
```

В структуре `MY_DEVICE_EXTENSION` определены две переменные: `Addr` и `Data`. Переменная `Addr` размером в два байта содержит адрес регистра устройства, к которому выполняются обращения, а байтовая переменная `Data` содержит данные, которые нужно записать в регистр `Addr` или прочитать их из него.

Напомню, что драйвер Windows представляет собой программный "контейнер", содержащий данные, структуры, функции обработки запросов с кодами IRP_MJ_xxx, а также функцию инициализации `DriverEntry`.

Вспомним, что каждый драйвер обрабатывает только те пакеты запросов, которые программист для него определил. В нашем случае драйвер должен обрабатывать запросы IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_DEVICE_CONTROL и Unload. Функция Unload выгружает драйвер из системы и обрабатывается несколько иным способом, чем пакеты запросов (в WDM-драйверах эта функция вообще не используется).

Как видно из исходного текста драйвера, здесь имеется функция `DriverEntry`, которая выполняет инициализацию драйвера при его запуске, и набор функций, которые обрабатывают запросы IRP, формируемые Менеджером ввода-вывода исполнительного модуля операционной системы. Наш драйвер должен обрабатывать следующие пакеты запросов:

- открытие устройства (функция с кодом IRP_MJ_CREATE);
- закрытие устройства (функция с кодом IRP_MJ_CLOSE);
- запись данных в устройство и чтение данных из устройства (функция с кодом IRP_MJ_DEVICE_CONTROL);
- удаление драйвера из системы (функция с кодом DriverUnload).

Проанализируем операции, которые выполняет каждая из этих функций, и начнем с функции IRP_MJ_CREATE, программный код которой приведен далее:

```
NTSTATUS
comdevice_Create(IN PDEVICE_OBJECT DeviceObject,
                 IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}
```

Первое, о чем нужно упомянуть — все функции-обработчики пакетов запросов принимают два параметра — адрес структуры DEVICE_OBJECT устройства и адрес пакета запроса PIRP. Оба параметра содержат поля, которые функция может использовать при обработке запроса. Наша функция-обработчик comdevice_Create, как и прочие функции обработки запросов, возвращает состояние выполнения запроса в переменной типа NTSTATUS. В этой функции выполняется несколько стандартных действий при выполнении запроса с кодом IRP_MJ_CREATE.

В поле Status пакета запроса заносится значение STATUS_SUCCESS, которое затем будет возвращено приложению как свидетельство успешно выполненной операции. Функция IoCompleteRequest завершает обработку пакета запроса. Сама функция возвращает значение STATUS_SUCCESS.

Точно так же работает и функция comdevice_Close, которая вызывается функцией WIN API CloseHandle() при закрытии дескриптора устройства в пользовательском приложении и соответствует запросу IRP_MJ_CLOSE.

Программный код функции comdevice_Unload показан далее:

```
VOID
comdevice_Unload (IN PDRIVER_OBJECT DriverObject)
{
   UNICODE_STRING DosDeviceName;
   RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
   IoDeleteSymbolicLink(&DosDeviceName);
   IoDeleteDevice(DriverObject->DeviceObject);
}
```

Функция выполняет операции, необходимые для полного удаления драйвера устройства из системы: удаляет символическую ссылку на устройство, о которой мы упоминали ранее (IoDeleteSymbolicLink()), и удаляет устройство из системы (IoDeleteDevice()).

Самая сложная по структуре функция comdevice_IOCTL, которая обрабатывает запрос IRP_MJ_DEVICE_CONTROL. В приложении пользователя этому запросу соответствуют вызовы функции DeviceIoControl() при записи данных в устройство и чтении данных из устройства. Для удобства работы в этой функции определен целый ряд переменных:

```
PMY_DEVICE_EXTENSION dx = (PMY_DEVICE_EXTENSION)DeviceObject-
>DeviceExtension;
```

```
PIO_STACK_LOCATION pIoStack = IoGetCurrentIrpStackLocation(Irp);
ULONG ctlCode = pIoStack->Parameters.DeviceIoControl.IoControlCode;

PUCHAR pBuf = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;
```

Переменная `dx` нужна для доступа к области расширения драйвера устройства, в которой будут храниться адрес `Addr` регистра UART и данные `Data` для записи или чтения из этого регистра. Переменная `pIoStack` позволяет получить доступ к области стека драйвера, в которую записывается различная информация, передаваемая вместе с запросом. Указатель `pIoStack` программа получает с помощью функции `IoGetCurrentIrpStackLocation`, параметром которой является адрес пакета запроса `Irp`.

В данном случае нас будет интересовать код команд записи-чтения IOCTL, который хранится в переменной `ctlCode`. Кроме того, для удобства работы мы используем переменную `pBuf`, указывающую на системный буфер данных, в котором Менеджер ввода-вывода будет хранить данные.

Далее в операторе `switch` анализируется IOCTL-код операции и выполняются соответствующие действия. Если код равен `IOCTL_WBYTE`, то выполняется запись байта данных в регистр, а если код равен `IOCTL_RBYTE` — то чтение. Для копирования данных из системной области в область расширения драйвера и наоборот используется функция `RtlMoveMemory`. Еще один важный момент: при чтении данных из устройства приложению должно возвращаться количество прочитанных байтов (поле `Information` структуры `Irp`). Операции чтения-записи выполняются с помощью функций `_inp` и `_outp` библиотеки C.

Функция `DriverEntry` вызывается один раз операционной системой при инициализации драйвера устройства. Функции передается ссылка на структуру `DriverObject`, созданную для нее операционной системой (первый параметр), и указатель на ключ данного драйвера в системном реестре (в записях этого ключа можно хранить или из записей извлекать при необходимости какие-то дополнительные параметры). Эта функция последовательно выполняет такие шаги (являются стандартными для этого класса драйверов):

1. Инициализируется константная строка с именем устройства в пространстве имен операционной системы (функция `RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME)`).
2. Создается программный объект описания устройства при помощи функции `IoCreateDevice()`. Если объект устройства создать не удалось, то функция заканчивает работу с ошибкой, код которой передается в переменной `status`, а сам объект устройства удаляется.

3. Инициализируется константная строка с именем устройства в пространстве имен, доступном пользовательской программе (функция `RtlInitUnicodeString (&DosDeviceName, DOS_DEVICE_NAME)`).
4. Создается символическая ссылка DOS-имени на имя из пространства имен операционной системы для доступа к устройству из приложения пользователя (функция `status = IoCreateSymbolicLink (&DosDeviceName, &NtDeviceName)`). В случае неудачи символическая ссылка и объект устройства удаляются, а функция `DriverEntry` заканчивается с ошибкой.
5. Устанавливаются флаги устройства.
6. Определяются функции-обработчики пакетов запроса.

В функции `DriverEntry` не следует пытаться выполнить какие-либо операции по вводу-выводу — программный код этой функции после выполнения удаляется системой из памяти.

Для компиляции и сборки драйвера устройства следует выполнить несколько шагов. Во-первых, в каталог, где находится файл с исходным текстом (в данном случае это comdriver.c), поместим еще два файла: makefile и sources.

Наличие этих файлов обязательно — с их помощью компилятор создаст (если нет ошибок в исходном тексте) файл драйвера устройства с именем comdriver.sys. Смысл записей в этих файлах я объяснять не буду — это все есть в документации. Вы можете просто взять любую пару этих файлов из каталогов, в которых находятся примеры, и изменить имя файла в sources следующим образом:

```
# The sources for the device driver:
TARGETNAME=comdriver
TARGETPATH=obj
TARGETTYPE=DRIVER
INCLUDES=..\
TARGETLIBS=  $(DDK_LIB_PATH)\wdmsec.lib
SOURCES=comdriver.c
```

После этого выберите консоль для запуска компилятора в среде Windows DDK, например, **Windows XP Checked Build Environment**, и перейдите в каталог, где находятся ваши рабочие файлы. Затем наберите команду

```
build -ceZ
```

Если в исходном тексте файла comdriver.c нет ошибок, то он будет откомпилирован успешно. Компилятор помещает файл драйвера comdriver.sys в отдельный каталог. В нашем случае это \objchk_wxp_x86\I386.

После компиляции нужно установить драйвер в систему и проверить его работу. Для установки (или удаления) драйвера воспользуемся программой *installer* (сайт **www.beyondlogic.org**) (рис. 3.18).

Рис. 3.18. Окно инсталляции драйвера

После успешной установки драйвера устройства разработаем тестовое приложение в среде Visual C++. Эта консольная программа будет передавать символ, введенный с клавиатуры консоли, посредством устройства UART программе-терминалу. Исходный текст программы передачи данных показан далее в листинге 3.11.

Листинг 3.11. Передача данных терминальной программе

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

#define IOCTL_WBYTE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                             0x801,\
                             METHOD_BUFFERED,\
                             FILE_ANY_ACCESS)
```

```
#define IOCTL_RBYTE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                             0x802,\
                             METHOD_BUFFERED,\
                             FILE_ANY_ACCESS)

#define PORT1    0x3F8

struct rwData {
       unsigned short Addr;
       char Byte;
  };
struct rwData mydata;
char dataWrite;

HANDLE fh;
DWORD bytes;

void WritePort(unsigned short Port, char data)
{
 mydata.Addr = Port;
 mydata.Byte = data;
 DeviceIoControl(fh,
                 IOCTL_WBYTE,
                 &mydata,
                 3,
                 NULL,
                 0,
                 &bytes,
                 NULL);
 return;
}

void main(void)
{
  fh = CreateFile("\\\\.\\comdevice",
```

```
                    GENERIC_READ | GENERIC_WRITE,
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);

  if (fh == INVALID_HANDLE_VALUE)
  {
    printf("Can't open device!\n");
    return;
  }
  printf("Enter chars, ESC to exit...\n");
  do
   {
    if (_kbhit())
     {
      dataWrite = _getch();
      WritePort(PORT1, dataWrite);
     }
   } while (dataWrite != 0x1B);
  CloseHandle(fh);
}
```

В начале программы следует определить коды IOCTL-команд, которые будут использоваться функцией `DeviceIoControl` при обращении к устройству `comdevice`. Эти команды (`IOCTL_WBYTE` и `IOCTL_RBYTE`) должны соответствовать тем, которые были определены в нашем драйвере comdriver.sys, иначе программа работать не будет.

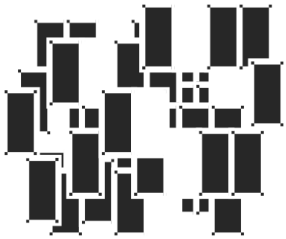
В программе также определяется структура `rwData`, содержащая поля `Addr` и `Byte` для хранения адреса регистра UART и байта данных. Адрес переменной `mydata` типа `rwData` будет передаваться в качестве параметра функции `DeviceIoControl` при записи данных в устройство UART. Функция `WritePort` записывает байт данных, переданный ей в качестве одного из параметров, по адресу, являющемуся вторым параметром. В программе опреде-

лена константа `PORT1`, которой присвоен базовый адрес порта COM1, равный 0x3F8.

Программа принимает символы, введенные с консоли, и отправляет их программе-терминалу, запущенной на этой же машине. Передача символов происходит до тех пор, пока в потоке введенных данных не попадется символ ESC. Чтобы получить доступ к устройству `comdevice`, используется функция WIN API `CreateFile()`.

При желании читатели могут расширить возможности этого приложения, добавив, например, программный код для чтения данных с программы-терминала. Напомню, что для проверки устройства UART выводы TxD и RxD последовательного порта должны быть соединены вместе.

Глава 4



Программирование последовательного интерфейса в операционных системах Windows

Последовательный порт компьютерной системы относится к коммуникационным ресурсам — физическим или логическим устройствам, обеспечивающим обмен данными посредством отдельного двунаправленного асинхронного потока данных.

Рассмотрим некоторые теоретические аспекты обработки данных в коммуникационных устройствах — они не только очень важны при анализе последовательного порта, но и позволяют понять основные принципы ввода-вывода в операционных системах Windows. Принципиально ввод-вывод данных может быть *синхронным* (блокирующим) или *асинхронным* (неблокирующим). При блокирующем вводе-выводе процесс ожидает завершения операции с данными, блокируя выполнение остальной части программного кода, при неблокирующем вводе-выводе обмен данными выполняется параллельно с программным кодом остальных частей программы.

Программная реализация асинхронного режима базируется на использовании нескольких программных технологий (использование потоков, ввода-вывода с перекрытием (overlapped) или портов завершения). С точки зрения производительности системы наиболее эффективным и простым в реализации является многопотоковый метод, когда для выполнения операций ввода-вывода используется отдельный программный поток или несколько потоков. При этом в каждом из потоков можно использовать блокирующие функции ввода-вывода, что упрощает программирование. Во всех последующих примерах программирования последовательного ввода-вывода будут использоваться один или несколько дополнительных потоков.

Для операционных систем Windows обмен данными по последовательному порту осуществляется посредством стандартных функций интерфейса Win API операционной системы для работы с объектами файловой системы. Прежде всего, замечу, что для программирования операций с COM-портом используются хорошо знакомые нам функции Win API `CreateFile`, `CloseHandle`, `ReadFile` и `WriteFile`, используемые при работе с файлами. Коммуникационные устройства в операционной системе обрабатываются теми же функциями, что и обычные файлы, при этом они имеют специальные имена, например, "COM1" присвоено первому последовательному порту, "COM2" — второму и т. д. Так, для открытия коммуникационного устройства COM1 можно использовать функцию `CreateFile`, первым параметром которой будет имя устройства, в данном случае, COM1. Полученный после вызова функции дескриптор устройства (handle) можно использовать при последующих операциях с данными в функциях `ReadFile`, `WriteFile` и `CloseHandle`.

Рассмотрим запись и чтение данных из COM-порта в операционных системах Windows на практических примерах. В качестве устройств последовательного обмена будем использовать персональный компьютер с имеющимся COM-портом, с одной стороны, и внешние устройства, в качестве которых будут выступать либо система на базе микроконтроллера 8051/8052, либо другой персональный компьютер, с другой стороны. Программное обеспечение для последовательного обмена данными можно протестировать и на одном персональном компьютере, используя один из двух вариантов. В первом можно соединить выводы RD и TD непосредственно на разъеме последовательного порта, а затем пробовать пересылать и принимать данные в одной и той же программе. Если имеется расширитель последовательного порта на базе PCI-модуля, то тестирование и настройку можно выполнить в полнофункциональном режиме. При этом одно приложение может записывать данные в один порт, а другое принимать эти данные на другом порту. Для такого тестирования следует вывод RD первого COM-порта соединить с выводом TD второго, и наоборот, вывод TD первого — с выводом RD второго.

4.1. Программирование последовательного ввода-вывода в C++ и Delphi

В первом примере выполняется запись строки байтов в последовательный порт COM1, к которому подключено внешнее устройство на микроконтроллере 8051/8052. В аппаратной реализации нашего проекта используется ин-

терфейс RS-232 с буфером MAX232. Нуль-модемное соединение использует 3 сигнальные линии: RxD, TxD и общий провод. Схема аппаратной части проекта показана на рис. 4.1.

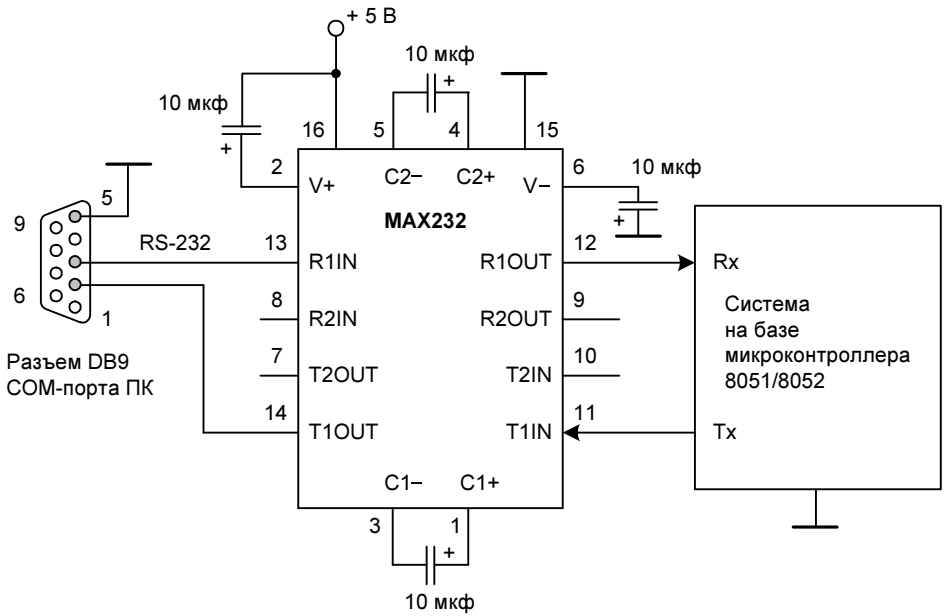


Рис. 4.1. Передача данных из последовательного порта во внешнее устройство с 8051/8052

Подобная схема будет использоваться в целом ряде последующих проектов, поэтому вкратце остановимся на схемотехническом решении. Здесь применяется драйвер интерфейса MAX232 для согласования уровней TTL-сигналов микроконтроллера 8051/8052 с выходом последовательного порта персонального компьютера. На выходе COM-порта имеется интегрированный буфер линии, поэтому для согласования уровней нужно устанавливать драйвер линии типа MAX232 или другой, что обычно делается на плате микроконтроллера. Как правило, все встроенные системы на базе микроконтроллеров или микропроцессоров включают такой драйвер линии или же реализуют его в отдельном интерфейсном кабеле. Для тестирования данного проекта использовалась плата разработки Rita-51 фирмы Rigel Corp. с установленным микроконтроллером 8052.

Программная часть проекта состоит из двух программ. Программа, прошитая в микроконтроллере, при получении символа с COM-порта ПК записывает полученный байт в порт P1. Программа, работающая под управлением Windows XP, каждые 5 сек инвертирует байт и отправляет его по интерфейсу RS-232 в микроконтроллерную систему.

Исходный текст консольного приложения, работающего с последовательным портом ПК, написан на языке C++ среды Micosoft Visual Studio 2005 и представлен в листинге 4.1.

Листинг 4.1. Win32-приложение для работы с COM-портом

```
#include <windows.h>
#include <stdio.h>

void main(void)
{
 HANDLE hCom;
 char *pcComPort = "COM1";
 DCB dcb;

 DWORD bytesWritten;
 char c1 = 0x0;

hCom = CreateFile(pcComPort,
                  GENERIC_READ | GENERIC_WRITE,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  0,
                  NULL);
if (hCom == INVALID_HANDLE_VALUE)
{
 printf("COM1 opening error!\n");
 return;
}
GetCommState(hCom, &dcb);
```

```
printf("COM1 baud rate is %d\n", dcb.BaudRate);
while (1)
{
  WriteFile(hCom,
            &c1,
            sizeof(c1),
            &bytesWritten,
            NULL);
  c1 = ~c1;
  Sleep(5000);
}
CloseHandle(hCom);
return;
}
```

Программа работает с портом COM1 персонального компьютера. Перед выполнением любых операций с устройством или файлом программа должна открыть это устройство и получить дескриптор (описатель) объекта, после чего все дальнейшие операции будут выполняться с дескриптором устройства. Устройство открывается с помощью функции

```
hCom = CreateFile(pcComPort,
                  GENERIC_READ | GENERIC_WRITE,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  0,
                  NULL);
```

При открытии коммуникационных портов в Windows необходимо указывать режим монопольного доступа, установив третий параметр функции `CreateFile` равным 0 и указав атрибут `OPEN_EXISTING`.

В случае успешного выполнения переменная `hCom` типа `HANDLE` будет хранить дескриптор объекта, который используется для записи данных в порт COM1. Функция `GetCommState()` применяется для получения информации об установленной скорости обмена по последовательному порту.

Далее программа в цикле `while (1)` каждые 5 сек записывает содержимое байтовой переменной `c1` в последовательный порт, после чего значение `c1` инвертируется. Перед завершением программы следует закрыть дескриптор устройства, вызвав функцию `CloseHandle()`.

Программа обработки данных, поступающих по последовательному интерфейсу микроконтроллера 8052, написана на языке Keil C51, и ее исходный текст представлен в листинге 4.2.

Листинг 4.2. Программа-обработчик данных последовательного интерфейса устройства 8052

```
#include <REG52.H>

void Serial_ISR(void) interrupt 4
{
 if (RI == 1)
   {
     RI = 0;
     P1 = SBUF;
   }
}

void main(void)
{
  P1 = 0x0;
  SCON = 0x50;
  TH1 = 0xF3 ;
  PCON |= 0x80;
  TMOD = 0x20;
  TR1 = 1;
  TI = 1;

  ES = 1;
  EA = 1;
  while (1);
 }
```

Микроконтроллер 8052 принимает байт данных с помощью функции-обработчика `Serial_ISR` прерывания последовательного порта, которая вызывается при установке флага приема RI при поступлении очередного байта данных в буфер приема `SBUF`. Значение полученного байта присваивается порту P1. Поскольку программа на компьютере пересылает инвертированный байт каждые 5 сек, то выходы порта P1 будут переключаться с таким же интервалом времени.

Приемник данных с последовательного порта настраивается на скорость 9600 бод.

Для тестирования этого проекта использовалась плата разработки с установленным микроконтроллером, работающим на частоте 24 МГц, поэтому Таймер 0 настраивается соответствующим образом. Инициализация прерывания последовательного порта выполняется операторами

```
ES = 1;
EA = 1;
```

Чтение данных в последовательный порт компьютера в блокирующем режиме выполняется с помощью функции Win API `ReadFile`. Для тестирования можно использовать схему аппаратной части, приведенную на рис. 4.1.

Исходный текст консольной программы на C++, работающей в операционной системе Windows XP персонального компьютера, приведен в листинге 4.3.

Листинг 4.3. Чтение данных с последовательного порта ПК

```
#include <windows.h>
#include <stdio.h>

void main(void)
{
 HANDLE hCom;
 char *pcComPort = "COM1";
 DCB dcb;

 DWORD bytesRead;
 char buf[128];
```

```
 hCom = CreateFile(pcComPort,
                   GENERIC_READ | GENERIC_WRITE,
                   0,
                   NULL,
                   OPEN_EXISTING,
                   0,
                   NULL);
if (hCom == INVALID_HANDLE_VALUE)
 {
  printf("COM1 opening error!\n");
  return;
 }

GetCommState(hCom, &dcb);
printf("COM1 baud rate is %d\n", dcb.BaudRate);
printf("Waiting for data from COM1...\n");
do {
    ReadFile(hCom,
             buf,
             sizeof(buf),
             &bytesRead,
             NULL);
    if (bytesRead > 0)
     {
      buf[bytesRead] = '\0';
      printf(buf);
      printf("\n");
    }
 }while (true);
CloseHandle(hCom);
return;
}
```

Как и в Win32-приложении из предыдущего примера, в этой программе вначале открывается файл последовательного порта COM1 с помощью функции Win API `CreateFile`.

В случае успешного открытия порта функция `CreateFile` возвращает дескриптор `hCom` коммуникационного порта, куда и будет записываться введенный с клавиатуры консоли символ. Функция `GetCommState` позволяет получить параметры последовательного порта, такие как скорость обмена, наличие бита четности и т. д. Полученные данные записываются в структуру `dcb` типа `DCB`. В данном случае для контроля скорости обмена данными последовательного порта персонального компьютера на экран консоли выводится значение переменной `dcb.BaudRate`. Думаю, излишне напоминать, что параметры последовательных портов устройств, обменивающихся данными, должны быть одинаковыми, иначе обмен данными будет невозможен.

Здесь, как и в предыдущем Windows-приложении, вначале открывается последовательный порт COM1 (Win API-функция `CreateFile`), затем строка из последовательного порта считывается в буфер `buf` функцией `ReadFile`. Если количество `bytesRead` принятых байтов не равно 0, то функция `printf` выводит принятую строку байтов на экран консоли ПК.

Исходный текст программы микроконтроллера 8052, которая посылает строку сообщения в последовательный порт ПК, показан в листинге 4.4.

Листинг 4.4. Программа-передатчик строки байтов в ПК

```
#include <REG52.H>

char *str = "Receiving string with RS-232";

void Serial_ISR(void) interrupt 4
{
 if (TI == 1)
   {
     TI = 0;
     if (*str != 0)
      SBUF = *str++;
   }
}

void main(void)
{
```

```
  SCON = 0x50;
  TH1 = 0xF3;
  PCON |= 0x80;
  TMOD = 0x20;
  TR1 = 1;
  TI = 1;

  ES = 1;
  EA = 1;
  while (1);
 }
```

Как и в программе для 8052 из предыдущего примера, обработка данных осуществляется в функции-обработчике `Serial_ISR` последовательного порта, только здесь анализируется установка флага передачи TI. Для инициализации начала передачи строки `str` в основной программе устанавливается флаг TI. При обнаружении нулевого символа передача данных прекращается.

Для программирования обмена данными в программе, написанной в среде Delphi, можно воспользоваться либо функциями Win API, либо включить в программу один из свободно распространяемых компонентов, выполняющих обработку данных с последовательного порта. Далее приводится исходный текст консольного приложения, разработанного в Delphi 2006 и принимающего данные с последовательного порта (листинг 4.5).

Листинг 4.5. Прием строки байтов с последовательного порта

```
program COM1_Receiver;

{$APPTYPE CONSOLE}

uses
  Windows,
  SysUtils;

var
 DevName: array[0..80] of Char;
 ComFile: THandle;
```

```
  buf: array[1..80] of Char;
  BytesRead: DWORD;
begin

  StrPCopy(DevName, 'COM1:');

  ComFile := CreateFile(DevName,
                        GENERIC_READ or GENERIC_WRITE,
                        0,
                        nil,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        0);

  if ComFile = INVALID_HANDLE_VALUE then
    Exit;
  while True do
   begin
     ReadFile(ComFile, buf, SizeOf(buf), BytesRead, nil);
     if BytesRead > 0 then
       WriteLn(buf);
   end;
  CloseHandle(ComFile);
end.
```

В этом приложении используются Win API-функции `CreateFile` и `ReadFile`. Для того чтобы эти функции работали, в программу нужно включить модуль Windows. Вначале программа открывает коммуникационное устройство COM1 с помощью функции `CreateFile`. При успешном выполнении функции в переменную `ComFile` типа `THandle` помещается дескриптор открытого устройства. Далее приложение в цикле `while` ожидает поступления строки байтов из последовательного порта, которые читаются функцией `ReadFile` в буфер `buf`. Если количество принятых байтов отлично от нуля, то содержимое `buf` выводится на экран консоли. По завершении работы дескриптор файла нужно закрыть функцией `CloseHandle`.

Передача байта данных в устройство по последовательному каналу выполняется по аналогии с листингом 4.1. Программный код на Delphi представлен в листинге 4.6.

Листинг 4.6. Передача байтов по последовательному каналу в программе на Delphi

```
program COM1_Transmitter;

{$APPTYPE CONSOLE}

uses
  Windows,
  SysUtils;

var
 DevName: array[0..80] of Char;
 ComFile: THandle;

  c1: BYTE;
  BytesWritten: DWORD;
begin

  StrPCopy(DevName, 'COM1:');
  c1:= 0;
  ComFile := CreateFile(DevName,
    GENERIC_READ or GENERIC_WRITE,
    0,
    nil,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    0);

  if ComFile = INVALID_HANDLE_VALUE then
    Exit;
  while True do
```

```
   begin
     WriteFile(ComFile, c1, SizeOf(c1), BytesWritten, nil);
     Sleep(10000);
     c1:= not c1;
   end;
  CloseHandle(ComFile);
end.
```

Как обычно, вначале последовательный порт открывается функцией `CreateFile`, а в переменную `ComFile` записывается дескриптор открытого файла. Запись байта `c1` выполняется в цикле `while` функцией `WriteFile`, после чего байт инвертируется. Цикл повторяется каждые 10 сек, и в микроконтроллер каждый раз передается инвертированный байт данных.

В состав интерфейса прикладного программирования включен и ряд функций, позволяющих выполнить настройку параметров канала последовательного обмена данными. В частности, очень полезной для разработчика является функция Win API `SetCommState`. Она позволяет настроить коммуникационный канал в соответствии со спецификациями, указанными в управляющем блоке устройства (структура типа `DCB`). С помощью функции `SetCommState` можно перенастроить оборудование без очистки входной или выходной очереди данных. Функция имеет следующий синтаксис:

```
BOOL SetCommState(HANDLE hFile,
                  LPDCB lpDCB);
```

Здесь первый параметр функции содержит дескриптор коммуникационного устройства, который должна возвращать функция `CreateFile`, а второй параметр — указатель на структуру `DCB`, поля которой содержат информацию данного коммуникационного устройства. При успешном завершении функции возвращаемое значение будет ненулевым, а при неудаче функция возвращает 0. В этом случае расширенную информацию о характере ошибки можно получить, вызвав функцию `GetLastError`.

Если программе необходимо поменять только некоторые параметры последовательного канала, то можно вызвать Win API-функцию `GetCommState`, которая возвращает заполненную структуру типа `DCB`. Далее можно установить требуемые параметры, оставив остальные без изменения. Функция `SetCommState` завершается с ошибкой, если поля `XonChar` и `XoffChar` структуры `DCB` имеют одинаковые значения.

Рассмотрим назначение отдельных полей структуры `DCB`:

- `DCBLength` — поле размером 32 бита. Определяет размер структуры в байтах;
- `BaudRate` — поле размером 32 бита. Определяет скорость обмена данными и может принимать одно из следующих значений (здесь число после символа подчеркивания указывает на скорость обмена в битах):
  - CBR_110
  - CBR_300
  - CBR_600
  - CBR_1200
  - CBR_2400
  - CBR_4800
  - CBR_9600
  - CBR_14400
  - CBR_19200
  - CBR_38400
  - CBR_57600
  - CBR_115200
  - CBR_128000
  - CBR_256000
- `fBinary` — поле размером 32 бита. Если значение поля равно TRUE, то устанавливается двоичный режим обмена данными (в операционных системах Windows никакой другой режим работы не поддерживается);
- `fParity` — поле размером 32 бита. Если значение этого поля установлено в TRUE, то выполняется проверка четности данных и генерируется отчет об ошибках;
- `fOutxCtsFlow` — поле размером 32 бита. Если значение этого поля установлено в TRUE, то будет отслеживаться состояние сигнала CTS (Clear-To-Send) при аппаратном контроле потока данных. Если при значении TRUE этого поля сигнал CTS сбрасывается, то передача данных приостанавливается до тех пор, пока CTS не будет вновь установлен в 1;
- `fOutX` — поле размером 32 бита. Указывает на необходимость использования программного (XON/XOFF) управления потоком данных при передаче данных. Если поле установлено в TRUE, то передача данных будет прекращена при получении символа `XoffChar` и возобновлена при получении символа `XonChar`;
- `fInX` — поле размером 32 бита. Указывает на необходимость использования программного (XON/XOFF) управления потоком данных при приеме данных. Если поле установлено в TRUE, то при переполнении буфера данных передатчику отправляется символ `XoffChar`, а при очистке буфера — символ `XonChar`;

- `fNull` — поле размером 32 бита. Если поле установлено в TRUE, то нулевой байт игнорируется при приеме данных;
- `fRtsControl` — поле размером 32 бита. Установка аппаратного контроля по сигналу RTS (Request-To-Send). Если поле установлено в `RTS_CONTROL_DISABLE`, то это запрещает установку сигнала на линии RTS, если поле установлено в `RTS_CONTROL_ENABLE` — это разрешает установку сигнала на линии RTS;
- `fAbortOnError` — поле размером 32 бита. Если значение этого поля установлено в TRUE, то при возникновении ошибок при операциях чтения/записи через последовательный порт все операции прекращаются. Дальнейшее продолжение операций возможно в том случае, если приложение отреагирует на ошибку и вызовет функцию `ClearCommError`;
- `XonLim` — поле размером 16 бит. Устанавливает минимальное число байтов в буфере приема, после превышения которого активизируется схема управления потоком данных, и передатчику посылается соответствующий сигнал. При этом передатчик может продолжать передавать байты данных, поэтому значение этого параметра никогда не станет равным 0. Использование этого параметра предполагает применение программного (XON/XOFF) или аппаратного управления потоком данных (RTS, DTR), параметры которого устанавливаются в `fInX`, `fRtsControl` или `fDtrControl`;
- `XoffLim` — поле размером 16 бит. Устанавливает максимальное число байтов во входном буфере до активизации схемы управления потоком данных при передаче. Использование этого параметра предполагает применение программного (XON/XOFF) или аппаратного управления потоком данных (RTS, DTR), параметры которого устанавливаются в `fInX`, `fRtsControl` или `fDtrControl`. При этом максимальное число байтов, которое можно передать, вычисляется как разность размера буфера и значения параметра `XoffLim`;
- `ByteSize` — поле размером 8 бит. Определяет количество битов в передаваемом или принимаемом байте;
- `Parity` — поле размером 8 бит. Устанавливает схему контроля четности (`EVENPARITY`, `ODDPARITY`, `NOPARITY`);
- `StopBits` — поле размером 8 бит. Определяет количество используемых стоповых битов (`ONESTOPBIT` (1 стоповый бит), `ONE5STOPBITS` (1,5 стоповых бита), `TWOSTOPBITS` (2 стоповых бита));

- `XonChar` — поле размером 8 бит. Значение символа `XON` при приеме и передаче данных;
- `XoffChar` — поле размером 8 бит. Значение символа `XOFF` при приеме и передаче данных;
- `EofChar` — поле размером 8 бит. Значение символа, который используется для указания конца данных;
- `EvtChar` — поле размером 8 бит. Значение символа, который используется для инициализации события.

Рассмотрим практическое применение функции конфигурирования последовательного интерфейса `SetCommState`. Следующий фрагмент программного кода на C++ позволяет установить скорость обмена, количество передаваемых битов и контроль четности:

```
HANDLE hCom;
BOOL fSuccess;
DCB dcb;

. . .

GetCommState(hCom, &dcb);

dcb.BaudRate = CBR_9600;
dcb.ByteSize = 8;
dcb.Parity = NOPARITY;
dcb.StopBits = ONESTOPBIT;

fSuccess = SetCommState(hCom, &dcb);

if (!fSuccess)
 {
   printf ("SetCommState failed with error %d.\n", GetLastError());
   return;
 }
```

Вначале с помощью функции `GetCommState` мы заполняем структуру `dcb` типа `DCB`. Затем устанавливаем соответствующие поля структуры `dcb`. Скорость обмена данными при новой установке будет равна 9600 бод, размер байта

данных 8 бит, без контроля четности и с одним стоповым битом. После сделанных изменений вызывается функция `SetCommState`, которая устанавливает новые параметры.

В следующем проекте рассмотрим работу простой программы, позволяющей одновременно передавать и принимать данные через последовательный порт. Аппаратная часть проекта включает последовательный порт COM1 персонального компьютера, на котором работает программа-эмулятор терминала, и последовательный COM-порт внешнего устройства. В качестве такого внешнего устройства может выступать другой персональный компьютер со свободным COM-портом, либо дополнительный COM-порт на одном и том же компьютере. Для разработки данного примера использовался PCI-модуль NetMos 9835 PCI Multi-I/O Controller с дополнительными COM-портами, установленный в свободный PCI-слот. Таким образом, тестирование проводилось на одном и том же ПК, на котором порты COM1 и COM4 были соединены нуль-модемным кабелем (рис. 4.2).

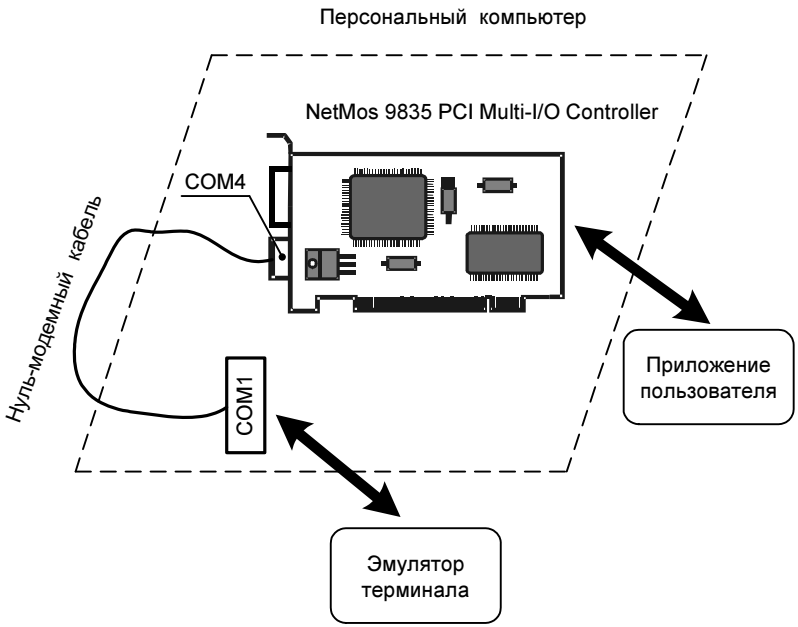


Рис. 4.2. Схема тестирования COM-портов

Программная часть проекта состоит из программы-эмулятора терминала, которая запущена на ПК и работает на порту COM1, и приложения пользовате-

ля, которое обрабатывает данные с порта COM4. В качестве терминальной программы можно взять одно из многочисленных свободно распространяемых приложений, установив соответствующим образом параметры обмена данными. В данной конфигурации используется программа Tera Term Pro, настроенная на работу с портом COM1 при скорости обмена 9600 бод, с 8-битовыми посылками, без проверки четности и одним стоповым битом.

Исходный текст программы пользователя для обмена данными между двумя COM-портами приведен в листинге 4.7.

Листинг 4.7. Программа пользователя, работающая с портом COM4

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

HANDLE hCom;
char c1;
DWORD bytesRead, bWritten;

BOOL fSuccess;
DWORD dwParam, dwThreadId;

VOID WINAPI ThreadProc(PVOID* dummy)
{
 printf("Waiting for data from COM4...\n");
 while (true)
  {
    ReadFile(hCom,
             &c1,
             sizeof(c1),
             &bytesRead,
             NULL);
   if (bytesRead > 0)
     printf("%c", c1);
   WriteFile(hCom,
             &c1,
```

```
            sizeof(c1),
            &bWritten,
            0);
  }
 }

void main(void)
{
  char *pcComPort = "COM4";
  DCB dcb;

  hCom = CreateFile(pcComPort,
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);
  if (hCom == INVALID_HANDLE_VALUE)
   {
    printf("COM4 opening error!\n");
    return;
   }

  GetCommState(hCom, &dcb);
  dcb.BaudRate = CBR_9600;
  dcb.ByteSize = 8;
  dcb.Parity = NOPARITY;
  dcb.StopBits = ONESTOPBIT;

  fSuccess = SetCommState(hCom, &dcb);
  if (!fSuccess)
   {
    printf ("SetCommState failed with error %d.\n", GetLastError());
    return;
   }
```

```
  printf ("Serial port %s successfully reconfigured.\n", pcComPort);
  GetCommState(hCom, &dcb);
  printf("COM4 baud rate is %d\n", dcb.BaudRate);

  hThread = CreateThread(NULL,
                         0,
                        (LPTHREAD_START_ROUTINE)ThreadProc,
                        &dwParam,
                        0,
                        &dwThreadId);

  while (1);
 }
  CloseHandle(hCom);
  return;
}
```

В этой программе обработка данных последовательного порта осуществляется в цикле `while` функции потока `ThreadProc`. Функция ожидает поступления данных через порт COM4. Данные читаются из порта побайтово функцией `ReadFile` в переменную `c1`. Если байт прочитан (переменная `bytesRead` содержит 1), то он выводится на экран консоли функцией `printf` и затем обратно передается в последовательный порт функцией `WriteFile`.

Конфигурирование порта COM4 выполняется в основной программе с помощью функции `SetCommState`. Для удобства вначале вызывается функция `GetCommState`, которая записывает параметры текущей конфигурации порта COM4 в структуру `dcb` типа `DCB`, затем в соответствующие поля этой структуры записываются нужные значения. Порт конфигурируется для работы со скоростью 9600 бод, с 8-битовой посылкой данных, без контроля четности и с одним стоповым битом. Информация о скорости обмена данными затем выводится на экран консоли.

Для работы с коммуникационным портом следует открыть его с помощью функции `CreateFile` и получить дескриптор для дальнейших манипуляций, что и делается в начале программы. Если функция завершается успешно, то в переменную `hCom` записывается значение дескриптора устройства. Для обработки данных, проходящих через порт COM4, программа запускает отдельный поток (функция `CreateThread`) с функцией потока `ThreadProc`. По за-

вершении работы программы следует закрыть дескриптор устройства функцией `CloseHandle`.

Вид окон работающих приложений пользователя и программы-эмулятора терминала показан на рис. 4.3.

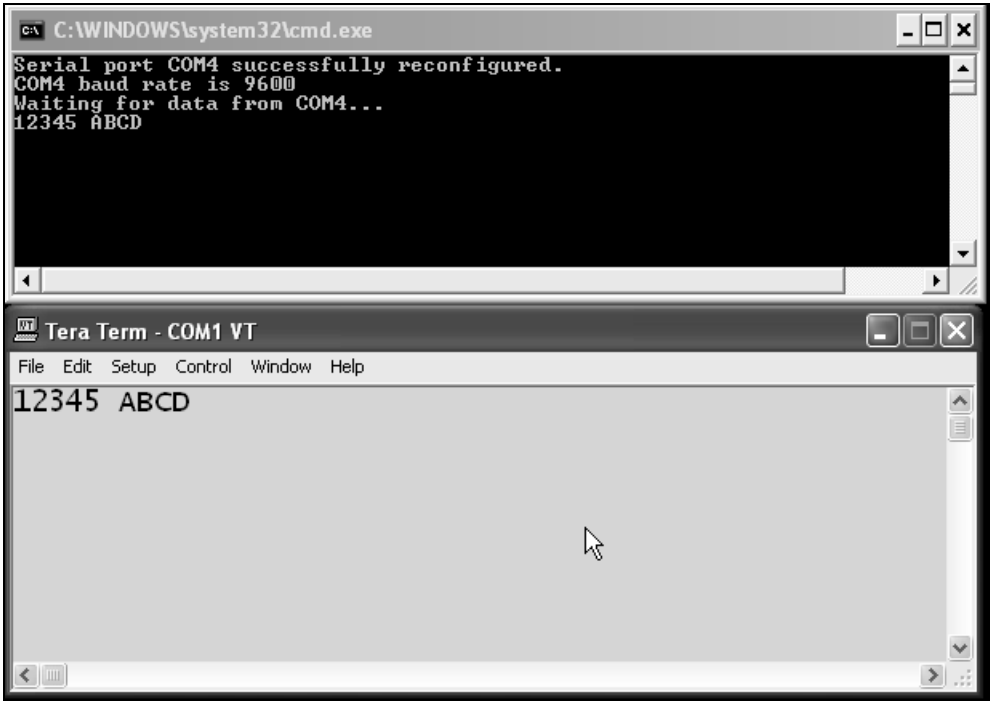


Рис. 4.3. Окна работающих приложений

Вместо платы расширения можно использовать дополнительный COM-порт (если имеется) на одном и том же ПК или же запустить терминальное приложение на одном компьютере, а тестовую программу — на другом. Если тестовая программа будет выполняться на отдельном компьютере, где имеется как минимум последовательный порт COM1, то в исходном тексте приложения следует заменить имя устройства с "COM4" на "COM1".

В структуре `DCB` определены поля, позволяющие разработчику программного обеспечения управлять потоком данных через последовательный порт аппаратным (RTS/CTS) или программным (XON/XOFF) способом. В следующем проекте демонстрируется программный метод управления потоком данных.

Здесь используется та же аппаратная конфигурация, что и в предыдущем проекте, а исходный текст тестового приложения показан в листинге 4.8. Демонстрационная программа подсчитывает количество байтов данных, отправленных программой-эмулятором терминала, и при поступлении 10 символов посылает терминалу символ останова (XOFF), после чего терминальная программа прекращает передачу данных. Тестовая программа "засыпает" на 20 сек, а затем посылает терминальной программе байт XOFF, разрешая передачу данных. Для демонстрации программного управления потоком в программе-эмуляторе терминала (в данном случае это Tera Term Pro) нужно установить опцию, разрешающую программное управление потоком данных.

Листинг 4.8. Программное (метод XON/XOFF) управление потоком данных в программе на C++

```
#include <windows.h>
#include <stdio.h>

#define XON 17
#define XOFF 19

char c1;
int counter = 0;

HANDLE hCom;
DWORD bytesRead, bWritten;

BOOL fSuccess;
HANDLE hThread, hThreadW;
DWORD dwParam, dwThreadId;

VOID WINAPI ThreadProc(PVOID* dummy)
{
 printf("Waiting for data from COM4...\n");
 while (true)
 {
   ReadFile(hCom,
            &c1,
```

```
            sizeof(c1),
            &bytesRead,
            NULL);
   if (bytesRead > 0)
      printf("%c", c1);
   counter++;
     if (counter > 10)
      {
       counter = 0;
       c1 = XOFF;
       WriteFile(hCom,
                 &c1,
                 sizeof(c1),
                 &bWritten,
                 0);
       Sleep(20000);
       c1 = XON;
       WriteFile(hCom,
                 &c1,
                 sizeof(c1),
                 &bWritten,
                 0);
       continue;
      }
   WriteFile(hCom,
             &c1,
             sizeof(c1),
             &bWritten,
             0);
   }
}

void main(void)
{
  char *pcComPort = "COM4";
  DCB dcb;
```

```
c1 = 0x0;
hCom = CreateFile(pcComPort,
                  GENERIC_READ | GENERIC_WRITE,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  0,
                  NULL);
if (hCom == INVALID_HANDLE_VALUE)
 {
  printf("COM4 opening error!\n");
  return;
 }

GetCommState(hCom, &dcb);
dcb.BaudRate = CBR_9600;
dcb.ByteSize = 8;
dcb.Parity = NOPARITY;
dcb.StopBits = ONESTOPBIT;
dcb.fOutX = TRUE;
dcb.XonChar = 17;
dcb.XoffChar = 19;

fSuccess = SetCommState(hCom, &dcb);
if (!fSuccess)
 {
  printf ("SetCommState failed with error %d.\n", GetLastError());
  return;
 }

printf ("Serial port %s successfully reconfigured.\n", pcComPort);
GetCommState(hCom, &dcb);
printf("COM4 baud rate is %d\n", dcb.BaudRate);
```

```
  hThread = CreateThread(NULL,
                          0,
                         (LPTHREAD_START_ROUTINE)ThreadProc,
                         &dwParam,
                         0,
                         &dwThreadId);

while (1);
CloseHandle(hCom);
return;
}
```

В данном приложении обмен данными осуществляется через последовательный порт COM4. При установке параметров последовательного порта помимо стандартных (скорость обмена, четность-нечетность и т. д.) в программе указано несколько дополнительных настроек. Первое — нужно разрешить использование программного управления (XON/XOFF) потоком данных, что выполняет оператор

```
dcb.fOutX = TRUE;
```

Затем нужно назначить коды байтам XON и XOFF. Общепринятые значения этих байтов равны 17 и 19 соответственно, хотя им можно присвоить и другие значения. Однако нестандартные коды XON и XOFF, используемые в конкретном приложении, могут вызвать проблемы при работе приложения с другими терминальными программами, в которых, как правило, используются стандартные кодировки этих символов. Установка параметров последовательного обмена данными выполняется посредством вызова функции `SetCommState`, которой передается дескриптор `hCom`, полученный при открытии порта функцией `CreateFile`, а также адрес структуры `dcb`.

Прием и анализ данных с порта COM4 осуществляется в отдельном потоке, который создается посредством вызова функции `CreateThread`. Функция `ThreadProc` этого потока выполняет несколько действий:

- читает байт данных с последовательного порта (функция `ReadFile`) и выводит байт, сохраненный в переменной `c1`, на экран консоли. Этот же байт отправляется обратно программе-терминалу;
- подсчитывает количество байтов данных с помощью переменной `counter` и при достижении значения 10 останавливает передачу данных програм-

мы-терминала, посылая байт XOFF с помощью функции `WriteFile`. Затем выполняется задержка в 20 сек, после чего разрешается дальнейшая передача данных посылкой байта XON терминальному приложению.

Более эффективным с точки зрения производительности является аппаратный метод управления потоком данных (метод RTS/CTS), показанный на рис. 4.4.

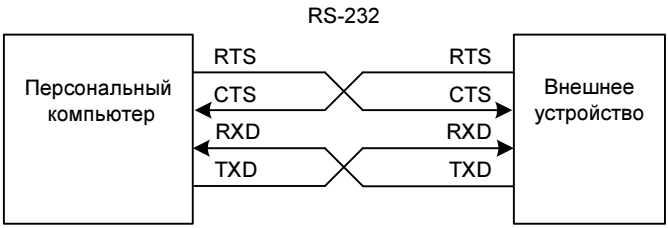


Рис. 4.4. Схема обмена данными с аппаратным управлением потоком

Здесь показано взаимодействие персонального компьютера с внешним устройством, хотя на обеих сторонах линии связи могут находиться как компьютерные системы, так и другие внешние устройства, например, модемы. При таком методе возможны несколько практических вариантов реализации управления потоком данных. Наиболее распространенный способ — указать устройству-передатчику на готовность приема данных с помощью установки активного уровня сигнала на его линии CTS. Для этого устройство-приемник устанавливает активный уровень сигнала на своей линии RTS. Получив такой сигнал, устройство-передатчик инициирует передачу данных, которая может быть прекращена при установке сигнала RTS приемника в неактивное состояние.

В операционных системах Windows аппаратное управление потоком данных можно выполнить с помощью функции Win API `EscapeCommFunction`. Функция имеет следующий синтаксис:

```
BOOL EscapeCommFunction (HANDLE hFile, DWORD dwFunc);
```

Здесь в качестве первого параметра (`hFile`) выступает дескриптор устройства, полученный ранее с помощью функции `CreateFile`, а в качестве второго — переменная, которая может принимать одно из следующих значений:

- `CLRDTR` — выполняет сброс сигнала DTR (Data-Terminal Ready);
- `CLRRTS` — выполняет сброс сигнала RTS (Request-To-Send);
- `SETDTR` — осуществляет установку сигнала DTR;

- `SETRTS` — осуществляет установку сигнала RTS;
- `SETXOFF` — эмулирует поведение системы при получении байта XOFF;
- `SETXON` — эмулирует поведение системы при получении байта XON;
- `SETBREAK` — приостанавливает передачу байтов данных и устанавливает линию в состояние "обрыва", пока не будет вызвана функция `ClearCommBreak`;
- `CLRBREAK` — возобновляет передачу данных и устанавливает линию в активное состояние.

В нашей программе будут использоваться две константы: `CLRRTS` — для останова передачи данных, `SETRTS` — для возобновления передачи. Как и в предыдущих примерах, мы будем использовать те же аппаратные средства. Кроме того, в нуль-модемном кабеле, соединяющем последовательные порты, должны быть задействованы линия RTS приемника и линия CTS передатчика — их нужно соединить.

Исходный текст демонстрационного приложения для аппаратного управления (метод RTS/CTS) показан в листинге 4.9.

Листинг 4.9. Аппаратное управление (метод RTS/CTS) потоком данных в программе на C++

```
#include <windows.h>
#include <stdio.h>

char c1;
int counter = 0;

HANDLE hCom;
DWORD bytesRead, bWritten;

BOOL fSuccess;
HANDLE hThread, hThreadW;
DWORD dwParam, dwThreadId;

VOID WINAPI ThreadProc(PVOID* dummy)
 {
  printf("Waiting for data from COM4...\n");
```

```
  while (true)
   {
     ReadFile(hCom,
              &c1,
              sizeof(c1),
              &bytesRead,
              NULL);
     if (bytesRead > 0)
      printf("%c", c1);
     counter++;
     if (counter > 7)
      {
       counter = 0;
       EscapeCommFunction(hCom, CLRRTS);
       Sleep(20000);
       EscapeCommFunction(hCom, SETRTS);
       continue;
      }
     WriteFile(hCom,
               &c1,
               sizeof(c1),
               &bWritten,
               0);
  }
}

void main(void)
{
  char *pcComPort = "COM4";
  DCB dcb;

  c1 = 0x0;
  hCom = CreateFile(pcComPort,
                    GENERIC_READ | GENERIC_WRITE,
                    0,
```

```
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);
if (hCom == INVALID_HANDLE_VALUE)
 {
  printf("COM4 opening error!\n");
  return;
 }

GetCommState(hCom, &dcb);
dcb.BaudRate = CBR_9600;     // установка скорости обмена 9600 бод
dcb.ByteSize = 8;             // установка формата данных 8 бит
dcb.Parity = NOPARITY;        // без бита четности
dcb.StopBits = ONESTOPBIT;    // один стоповый бит
dcb.fRtsControl = RTS_CONTROL_ENABLE;

fSuccess = SetCommState(hCom, &dcb);

if (!fSuccess)
 {
  printf ("SetCommState failed with error %d.\n", GetLastError());
  return;
 }

printf ("Serial port %s successfully reconfigured.\n", pcComPort);
GetCommState(hCom, &dcb);
printf("COM4 baud rate is %d\n", dcb.BaudRate);

hThread = CreateThread(NULL,
                       0,
                       (LPTHREAD_START_ROUTINE)ThreadProc,
                       &dwParam,
                       0,
                       &dwThreadId);
```

```
  while (1);
  CloseHandle(hCom);
  return;
}
```

Основные функции по обработке данных выполняются в отдельном потоке с помощью функции `ThreadProc`. Функция принимает байты данных с последовательного порта COM4 от терминального приложения и инкрементирует счетчик `counter`. Если принятых символов оказывается больше семи, то сигнал на линии RTS приемника сбрасывается в неактивное состояние функцией `EscapeCommFunction`:

```
EscapeCommFunction(hCom, CLRRTS);
```

Передатчик данных (терминальная программа), получив такой сигнал, приостанавливает передачу данных приемнику. Функция потока тестового приложения ожидает 20 сек, а затем устанавливает сигнал на линии RTS в активное состояние, разрешая передатчику передавать данные:

```
Sleep(20000);
EscapeCommFunction(hCom, SETRTS);
```

Для применения аппаратного управления потоком данных поле `fRtsControl` структуры `dcb` должно быть установлено соответствующим образом, что в основной программе выполняет оператор:

```
dcb.fRtsControl = RTS_CONTROL_ENABLE;
```

Смысл остальных операторов демонстрационного приложения понятен из предыдущих примеров, поэтому останавливаться на них я не буду.

В следующем примере показан один из возможных вариантов сохранения данных, полученных с последовательного порта, в текстовом файле. Аппаратная часть проекта такая же, как и в предыдущих примерах, а исходный текст программы показан в листинге 4.10.

Листинг 4.10. Запись данных последовательного порта в файл в программе на C++

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
```

```
char buf[128];
char *pbuf = buf;
int counter = 0;

HANDLE hCom, hEvent;
DWORD bytesRead, bWritten;

BOOL fSuccess;
HANDLE hThread;
DWORD dwParam, dwThreadId;

VOID WINAPI ThreadProc(PVOID* dummy)
{
 printf("Waiting for data from COM4...\n");
 while (true)
 {
     ReadFile(hCom,
              pbuf,
              1,
              &bytesRead,
              NULL);
     if (bytesRead > 0)
      {
       WriteFile(hCom,
                 pbuf++,
                 1,
                 &bWritten,
                 NULL);
                 counter++;
      if (counter == 10)
       {
        SetEvent(hEvent);
        buf[counter] = '\0';
        counter = 0;
```

```
        pbuf = buf;
       }
    }
  }
}

void main(void)
{
  char *pcComPort = "COM4";
  DCB dcb;
  FILE *fout;

  hCom = CreateFile(pcComPort,
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);
  if (hCom == INVALID_HANDLE_VALUE)
   {
     printf("COM4 opening error!\n");
     return;
   }

  GetCommState(hCom, &dcb);
  dcb.BaudRate = CBR_9600;
  dcb.ByteSize = 8;
  dcb.Parity = NOPARITY;
  dcb.StopBits = ONESTOPBIT;
  fSuccess = SetCommState(hCom, &dcb);

  if (!fSuccess)
   {
    printf ("SetCommState failed with error %d.\n", GetLastError());
    return;
   }
```

```
  printf ("Serial port %s successfully reconfigured.\n", pcComPort);
  GetCommState(hCom, &dcb);
  printf("COM4 baud rate is %d\n", dcb.BaudRate);

  hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  hThread = CreateThread(NULL,
                         0,
                         (LPTHREAD_START_ROUTINE)ThreadProc,
                         &dwParam,
                         0,
                         &dwThreadId);
  if ((fout = fopen("c:\\Test", "a+")) == NULL)
   {
    printf("Can't create file!\n");
    return;
   }
  printf("Press Enter to exit...\n");
  do
    {
      WaitForSingleObject(hEvent, INFINITE);
      printf("%s\n", buf);
      fwrite(buf, sizeof(char), 10, fout);
    } while (!kbhit());

  fclose(fout);
  CloseHandle(hCom);
  return;
}
```

В этой программе каждые 10 байтов данных, прочитанных с последовательного порта COM4 функцией потока `ThreadProc`, записываются в текстовый файл с именем Test на диске C. Для синхронизации записи данных в файл и приема байтов с последовательного порта используется функция Win API `WaitForSingleObject`, которая отслеживает состояние объекта события с дескриптором `hEvent`.

Как только объект "событие" переходит в сигнальное состояние, которое устанавливается функцией

```
SetEvent(hEvent);
```

в функции потока, функция ожидания `WaitForSingleObject` в главном потоке завершается, и содержимое буфера данных выводится на консоль и записывается в дескриптор `fout` файла Test. Как и в предыдущих примерах, параметры последовательного порта устанавливаются с помощью функции `SetCommState`. Для создания и работы с файлом Test используются библиотечные функции C++ `fopen`, `fwrite` и `fwclose`. Объект "событие", используемый для синхронизации основного и дополнительного потока программы, создается функцией

```
hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

Полученный дескриптор объекта сохраняется в переменной `hEvent`. "Событие" создается в несигнальном состоянии и сбрасывается автоматически после перехода в сигнальное состояние.

Данные с последовательного порта COM4 записываются в буфер `buf` функцией `ReadFile` в функции потока `ThreadProc`. Для удобства адресации данных в буфере используется указатель `pbuf`. В переменной `counter` хранится количество байтов данных, полученных с последовательного порта. Как только это количество достигает 10, объект "событие" с дескриптором `hEvent` переводится в сигнальное состояние.

Выход из цикла записи данных `while` основной программы происходит при нажатии клавиши <Enter>, после чего дескрипторы коммуникационного канала `hCom` и текстового файла `fout` закрываются.

Рассмотренные нами примеры представляют собой консольные приложения, в которых обработка данных с последовательного порта осуществляется с помощью нескольких функций прикладного интерфейса программирования, предоставляемых операционными системами Windows (`SetCommState`, `GetCommState` и `EscapeCommFunction`). Для более детальной настройки последовательного порта и получения полного контроля над процессом обмена данными имеется целый ряд других функций Win API, которые доступны программисту (`SetCommTimouts`, `GetCommTimeouts`, `WaitCommEvent` и т. д.). Эти функции достаточно подробно описаны в документации Microsoft, поэтому читатели, желающие более досконально изучить программирование последовательного порта, могут обнаружить здесь всю интересующую их информацию.

При разработке приложений с графическим интерфейсом пользователя в среде Delphi и Microsoft Visual Studio .NET версий 2005 и 2008 для программирования последовательного обмена данными очень удобно использовать визуальные компоненты. Компоненты инкапсулируют функции прикладного интерфейса программирования для последовательного порта, что существенно упрощает программирование графических приложений.

Так, в последние версии Visual Studio .NET (2005 и 2008) включен компонент SerialPort, позволяющий автоматизировать многие настройки последовательного порта. Что же касается среды Delphi, то в базовом пакете отсутствуют визуальные компоненты для работы с последовательным портом, зато имеется несколько очень популярных графических компонентов, свободно распространяемых в Интернете и доступных для закачки. К таким компонентам относятся *TComPort* и программный пакет *TurboPower Async Professional*, включающий, кроме последовательного порта, целый ряд визуальных компонентов для работы в сети. В данной книге для демонстрационных приложений используется пакет TurboPower AsyncPro 4.07.

Следующий раздел этой главы посвящен программированию обмена данными по последовательному порту с применением графических компонентов SerialPort (Visual Studio .NET 2005) и AsyncPro (Delphi 2006/2007). Для тестирования приложений будет использоваться персональный компьютер с установленной PCI-платой расширителя интерфейса NetMos 9835 PCI Multi-I/O Controller. Последовательный порт COM1 ПК соединяется с портом COM4 платы расширителя. На порту COM1 работает терминальная программа, а тестовые приложения передают/получают данные через порт COM4 платы расширителя. Порты COM1 и COM4 соединяются между собой нуль-модемным кабелем.

4.2. Программирование последовательного ввода-вывода в Delphi

При правильной инсталляции в Delphi закладка с компонентами AsyncPro должна появиться в палитре компонентов (рис. 4.5) при разработке нашего первого графического приложения.

В нашем приложении будет использоваться компонент TApdComPort, с помощью которого текстовая строка будет передана через порт COM4. Пере-

тащим компонент на форму приложения и установим параметры последовательного обмена данными (рис. 4.6).

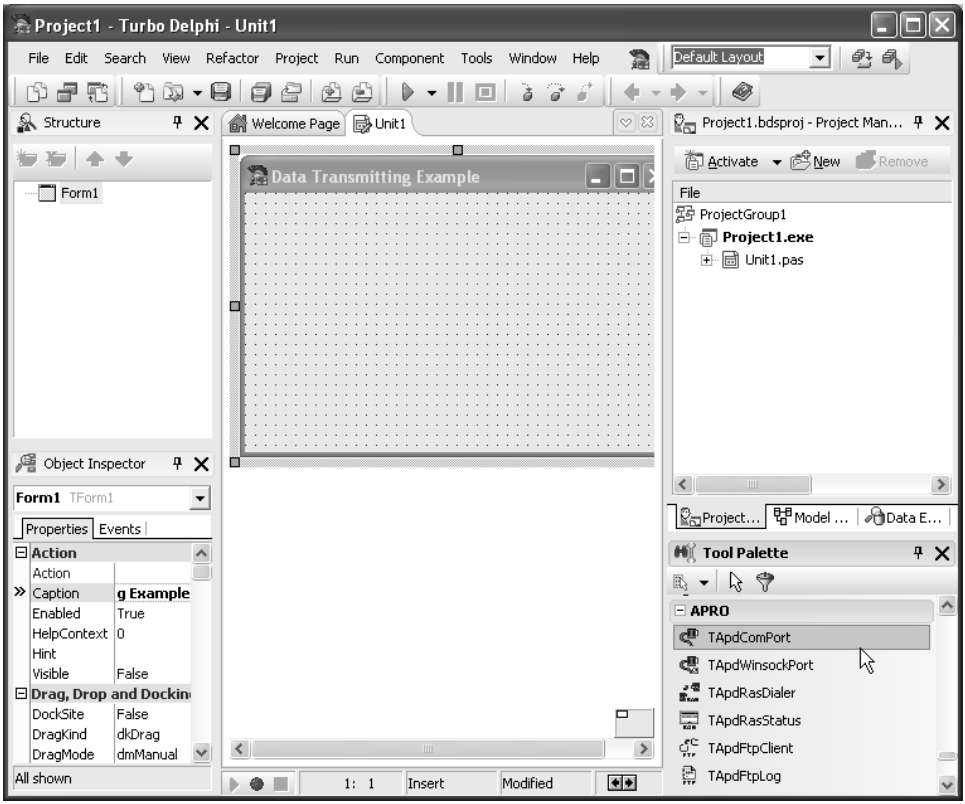


Рис. 4.5. Размещение компонентов AsyncPro в палитре компонентов

В окне свойств экземпляра компонента ApdComPort1 установим скорость обмена **Baud** равной 9600 бод, а **ComNumber** (номер порта) — равным 4. Остальные параметры оставим без изменения.

Поместим на форму кнопку из группы стандартных компонентов и установим заголовок **Send String to COM4** (рис. 4.7). Функция-обработчик нажатия этой кнопки будет передавать строку текста в порт COM4.

Строка текста будет передаваться при нажатии на кнопку, поэтому напишем код функции-обработчика нажатия кнопки. Исходный текст этой функции представлен листингом 4.11.

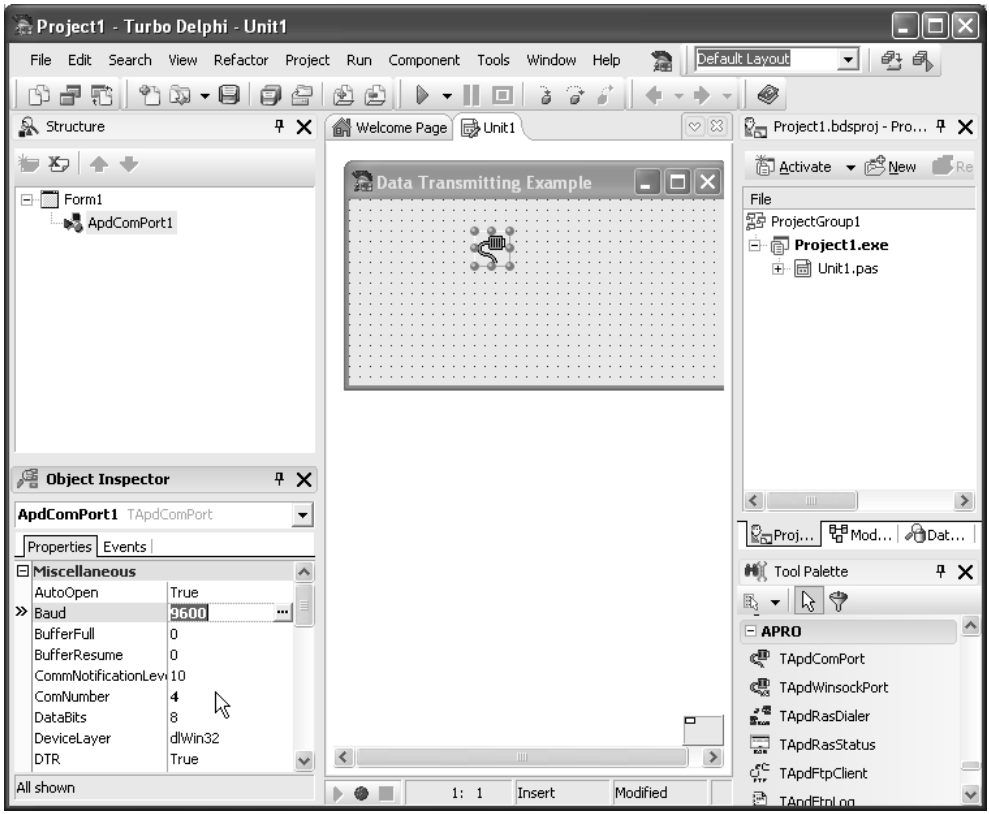


Рис. 4.6. Установка параметров последовательного обмена

Листинг 4.11. Функция-обработчик нажатия кнопки *Send String to COM4*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 ApdComPort1.Output:= 'Test String for COM4';
end;
```

Для передачи строки через порт COM4 нужно присвоить ее значение свойству `Output` компонента ApdComPort1. Поскольку мы оставили свойство `AutoOpen` компонента установленным по умолчанию в True, то открывать порт COM4 вручную не нужно.

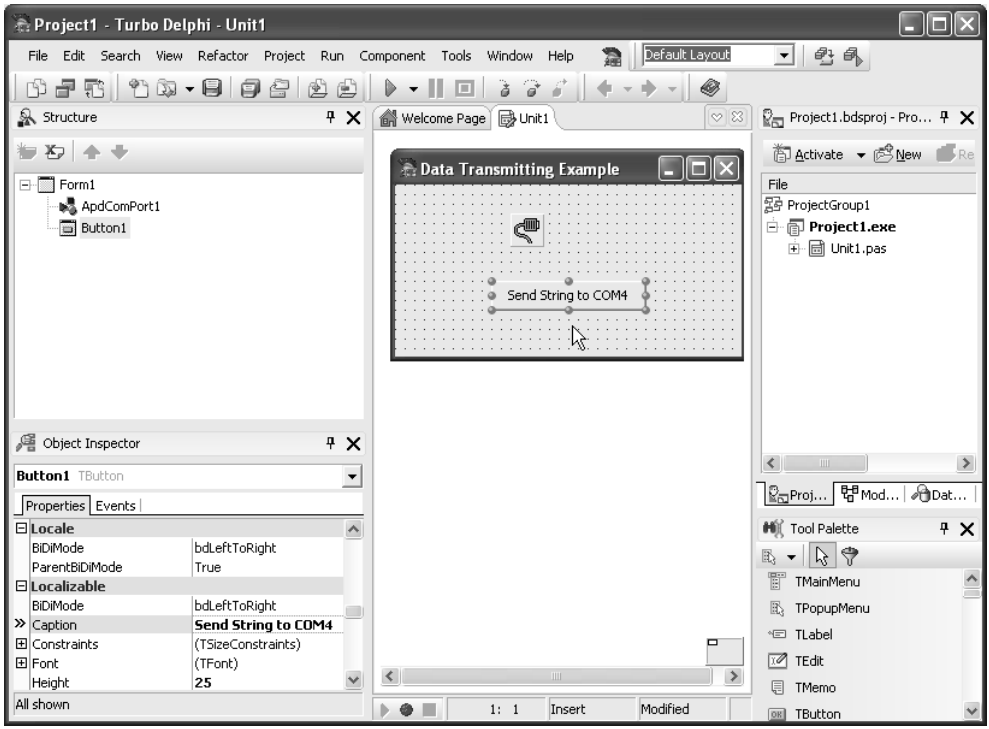


Рис. 4.7. Установка компонента Button

После компиляции и сборки приложения проверим работоспособность программы, для этого запустим программу-эмулятор терминала, которая будет принимать данные приложения на порту COM1. При нажатии кнопки **Send String to COM4** текстовая строка отобразится в окне терминала (рис. 4.8).

Для приема данных из COM-порта в компоненте ApdComPort можно настроить один из обработчиков событий. Событие происходит, например, при поступлении данных в буфер ввода последовательного порта, при этом автоматически вызывается функция-обработчик этого события.

Разработаем тестовое приложение, которое будет принимать данные с последовательного порта COM4 и отображать в окне текстового редактора. Для этого поместим на форму разрабатываемого приложения экземпляр компонента TApdComPort и однострочного текстового редактора TEdit (рис. 4.9).

Установим основные свойства компонента ApdComPort1 такими же, как и в предыдущем примере (скорость 9600 бод, 8-битовая посылка, номер порта 4), и отключим свойство `AutoOpen`, установив его в False. Компонент TApdCom-

Port позволяет отслеживать появление одного из нескольких событий. В данном случае нужно отслеживать появление данных в буфере приема, поэтому установим функцию-обработчик для события **OnTriggerAvail** (рис. 4.10).

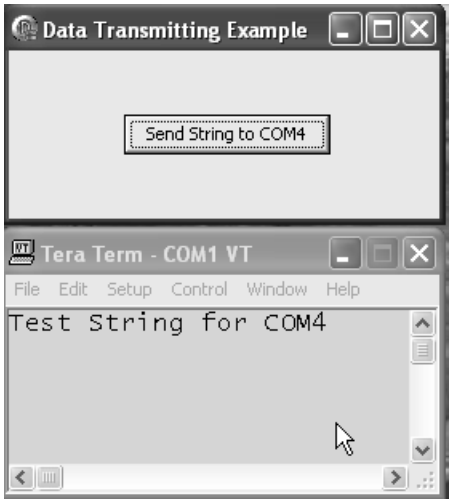


Рис. 4.8. Тестирование приложения

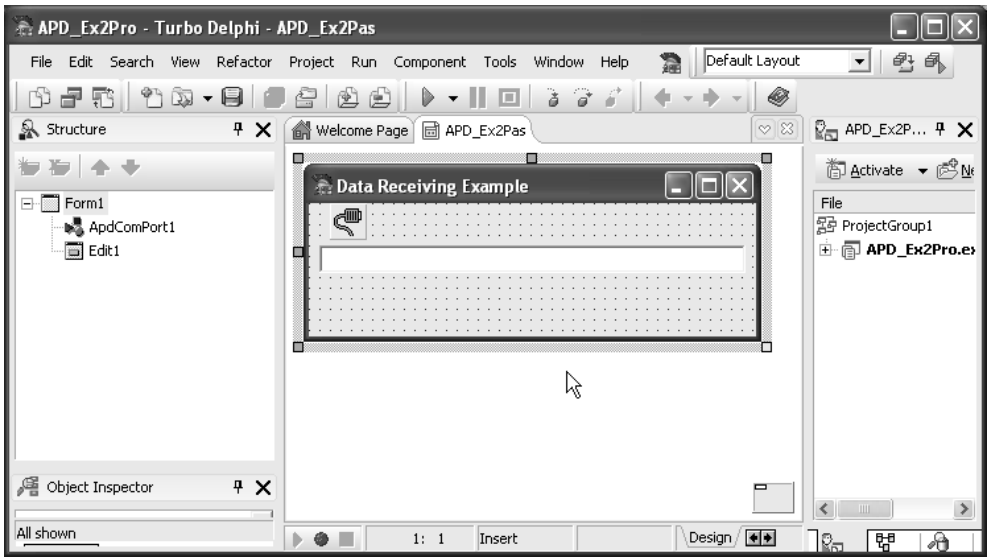


Рис. 4.9. Окно конструктора приложения

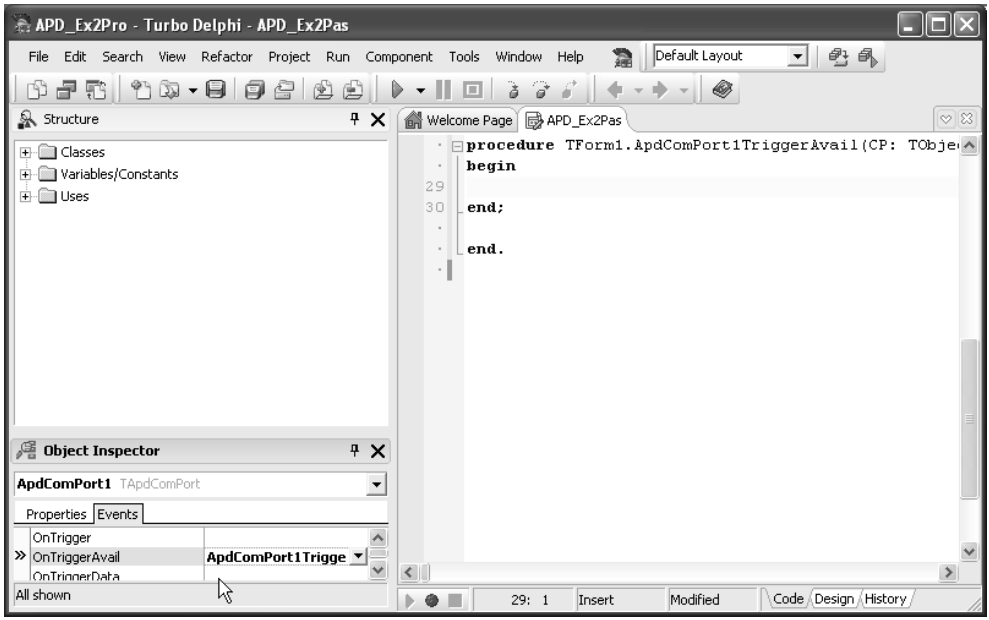


Рис. 4.10. Установка обработчика события компонента ApdComPort1

Функция-обработчик события `OnTriggerAvail` должна отобразить полученные с порта COM4 данные в окне текстового редактора, для чего используется следующий исходный текст (листинг 4.12).

Листинг 4.12. Функция-обработчик для приема данных через последовательный порт

```
procedure TForm1.ApdComPort1TriggerAvail(CP: TObject; Count: Word);
var
 I: WORD;
 C: Char;
 S: String;
begin
 for I := 1  to Count do
   begin
     C:= ApdComPort1.GetChar();
     S:= S + C;
   end;
```

```
   Edit1.Text:= Edit1.Text + S;
end;
```

По умолчанию функции-обработчику события `OnTriggerAvail` присваивается имя `ApdComPort1TriggerAvail`. Параметр `Count` функции содержит счетчик полученных байтов. Находящийся в буфере приемника очередной байт данных считывается в переменную `C` функцией `GetChar`, а все полученные байты данных накапливаются в строке `S`, содержимое которой выводится в окно однострочного редактора Edit1.

Кроме того, при запуске приложения нужно открыть последовательный порт, а при завершении — закрыть его. Полный исходный текст приложения показан в листинге 4.13.

Листинг 4.13. Исходный текст приложения, принимающего данные с порта COM4

```
unit APD_Ex2Pas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, OoMisc, AdPort;

type
  TForm1 = class(TForm)
  ApdComPort1: TApdComPort;
  Edit1: TEdit;
  procedure ApdComPort1TriggerAvail(CP: TObject; Count: Word);
  procedure FormCreate(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.ApdComPort1TriggerAvail(CP: TObject; Count: Word);
var
 I: WORD;
 C: Char;
 S: String;
begin
 for I := 1  to Count do
   begin
     C:= ApdComPort1.GetChar();
     S:= S + C;
     ApdComPort1.Output:= S;
   end;
   Edit1.Text:= Edit1.Text + S;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 ApdComPort1.Open:= False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 ApdComPort1.Open:= True;
end;
end.
```

Исходный текст содержит функцию-обработчик события открытия формы приложения (`FormCreate`), в которой порт COM4 открывается присвоением свойству `Open` класса `ApdComPort1` значения True. При закрытии приложения

(обработчик `FormClose`) порт закрывается присвоением свойству `Open` значения False.

Для тестирования приложения нужно запустить терминальную программу на порту COM1, а нашу программу — на порту COM4. Введенные в окне терминала символы будут выводиться в окно редактирования приложения.

До сих пор при создании приложений с компонентом TApdComPort мы не касались вопросов управления потоком данных. Следующий программный проект демонстрирует программный метод (XON/XOFF) управления потоком данных через последовательный порт. На форме приложения разместим экземпляры компонентов TApdComPort, однострочного редактора TEdit и двух кнопок TButton (рис. 4.11).

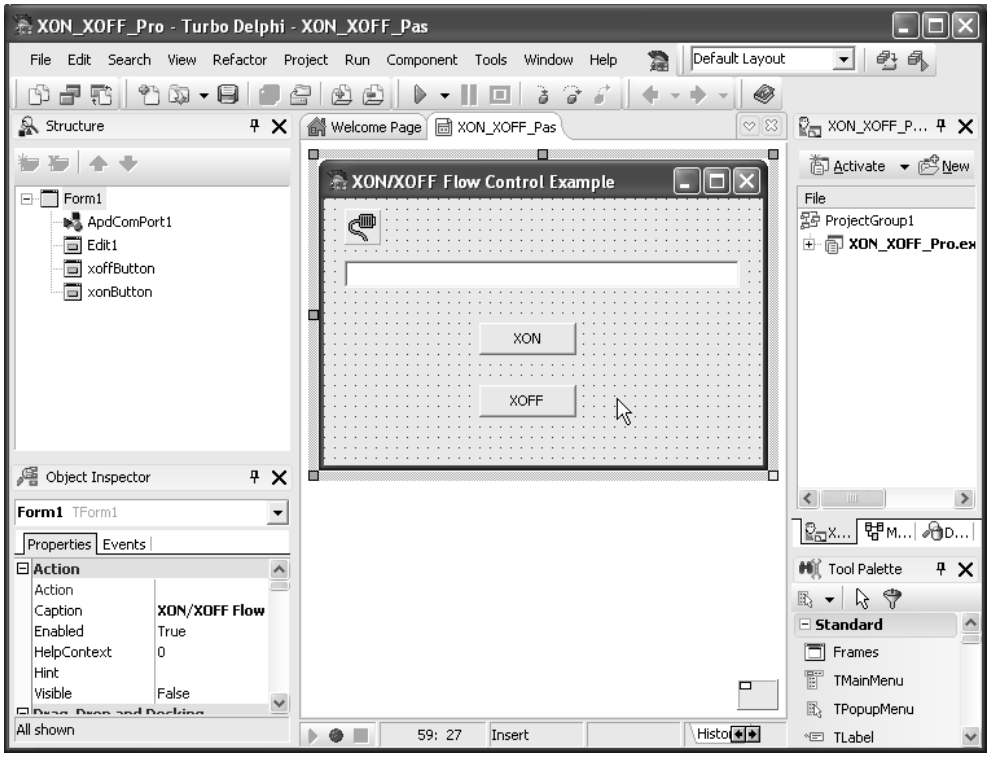


Рис. 4.11. Вид окна конструктора приложения

Одну из кнопок назовем xonButton и присвоим ей заголовок **XON**, другую назовем xoffButton с заголовком **XOFF**. При нажатии кнопки **XON** в порт

COM4 будет передан байт XON, а при нажатии кнопки **XOFF** — байт XOFF. Программа-эмулятор терминала, работающая на порту COM1, при получении байта XOFF приостановит передачу данных, а при получении символа XON возобновит ее.

В свойствах компонента ApdComPort1 должны быть установлены стандартные для протокола XON/XOFF значения (рис. 4.12).

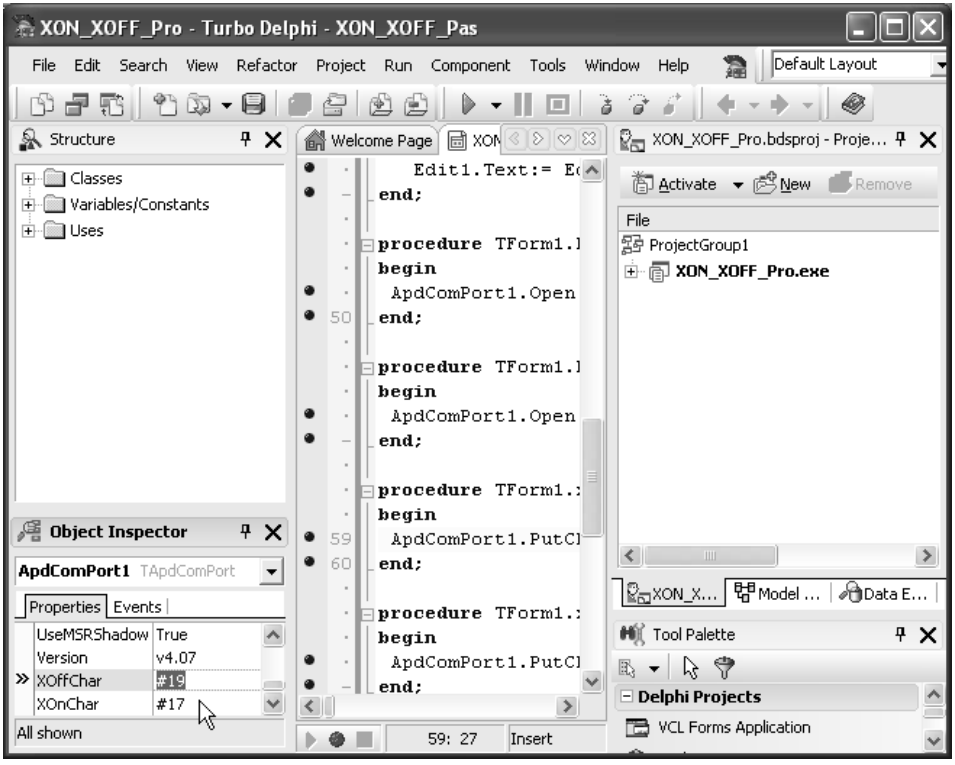


Рис. 4.12. Установки параметров протокола XON/XOFF

Параметр `AutoOpen` компонента ApdComPort1должен быть установлен в False, а остальные параметры имеют стандартные значения, как и в предыдущих примерах. Приложение будет принимать символы с последовательного порта COM4 и отображать их в окне Edit1. При нажатии кнопки **XOFF** прием символов остановится из-за остановки передатчика, а при нажатии **XON** прием возобновляется. Исходный текст приложения приведен в листинге 4.14.

Листинг 4.14. Программа приема данных с программным (XON/XOFF) управлением потоком

```
unit XON_XOFF_Pas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, OoMisc, AdPort;

type
  TForm1 = class(TForm)
  ApdComPort1: TApdComPort;
  Edit1: TEdit;
  xonButton: TButton;
  xoffButton: TButton;
  procedure ApdComPort1TriggerAvail(CP: TObject; Count: Word);
  procedure FormCreate(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  procedure xonButtonClick(Sender: TObject);
  procedure xoffButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.ApdComPort1TriggerAvail(CP: TObject; Count: Word);
var
 I: WORD;
```

```
 C: Char;
 S: String;
begin
 for I := 1 to Count do
   begin
    C:= ApdComPort1.GetChar;
    S:= S + C;
   end;
   Edit1.Text:= Edit1.Text + S;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 ApdComPort1.Open:= False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 ApdComPort1.Open:= True;
end;

procedure TForm1.xoffButtonClick(Sender: TObject);
begin
 ApdComPort1.PutChar(#19);
end;

procedure TForm1.xonButtonClick(Sender: TObject);
begin
 ApdComPort1.PutChar(#17);
end;

end.
```

Многие фрагменты кода нам уже знакомы из предыдущих примеров, поэтому я остановлюсь лишь на обработчиках нажатий кнопок. Функция-обработчик нажатия кнопки **XON** `xonButtonClick` выполняет передачу сим-

вола XON в последовательный порт посредством функции `PutChar`. Эта же функция используется и в обработчике нажатия кнопки `xoffButtonClick`, но здесь в порт COM4 передается символ XOFF. Для тестирования приложения необходимо, чтобы в терминальной программе, работающей на порту COM1, была включена опция XON/XOF (она может называться по-другому в разных программах-эмуляторах, но суть остается прежней).

Следующий пример применения компонента TApdComPort, который мы рассмотрим — аппаратное управление (метод RTS/CTS) потоком данных. Для тестирования этого метода необходимо, чтобы сигнальные выводы RTS и CTS нуль-модемного кабеля были попарно соединены. Наше приложение будет останавливать или возобновлять передачу данных терминальной программой при установке или сбросе сигнала на линии RTS. Терминальная программа при установленной опции аппаратного управления перед отправкой очередного байта проверяет уровень сигнала на линии CTS, соединенной с линией RTS приемника. При высоком (активном) уровне сигнала на линии CTS программа-эмулятор отправляет очередной байт данных приемнику, при низком (неактивном) уровне сигнала данные помещаются в буфер и остаются там до активизации сигнала RTS приемника.

Вид окна конструктора тестового приложения показан на рис. 4.13.

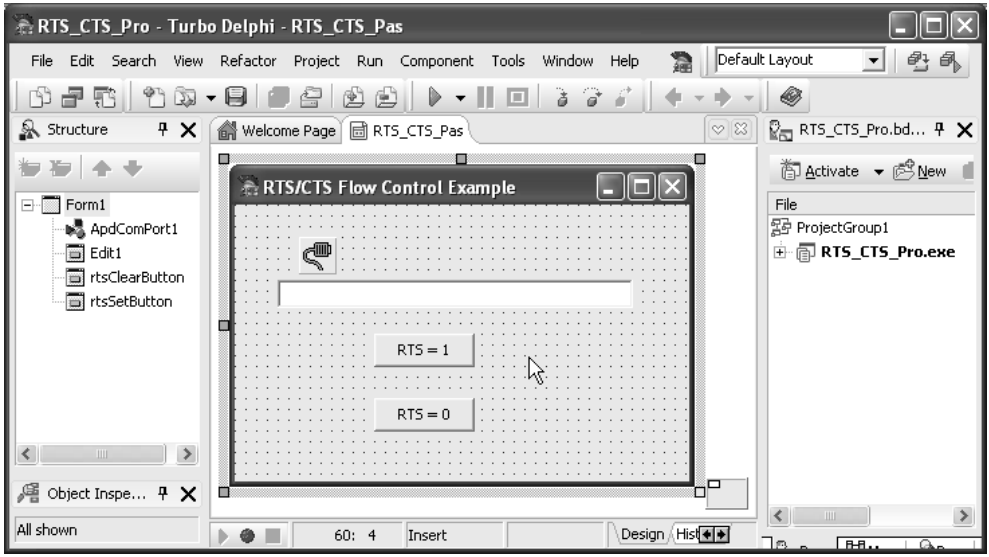


Рис. 4.13. Окно конструктора приложения

На форму приложения мы помещаем экземпляр компонента TApdComPort, однострочный текстовый редактор Edit1 и две кнопки с именами rtsSetButton (заголовок **RTS = 1**) и rtsClearButton (заголовок **RTS = 0**). При нажатии кнопки **RTS = 1** (rtsSetButton) на линии RTS будет установлен высокий (активный) уровень напряжения, который разрешает передачу байтов данных приемнику. Если нажата кнопка **RTS = 0** (rtsClearButton), то сигнал RTS становится неактивным, и передача данных приостанавливается.

Исходный текст демонстрационной программы со всеми обработчиками показан в листинге 4.15.

Листинг 4.15. Программа приема данных с аппаратным (RTS/CTS) управлением потоком

```
unit RTS_CTS_Pas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, OoMisc, AdPort;

type
  TForm1 = class(TForm)
  ApdComPort1: TApdComPort;
  Edit1: TEdit;
  rtsSetButton: TButton;
  rtsClearButton: TButton;
  procedure ApdComPort1TriggerAvail(CP: TObject; Count: Word);
  procedure FormCreate(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  procedure rtsSetButtonClick(Sender: TObject);
  procedure rtsClearButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.ApdComPort1TriggerAvail(CP: TObject; Count: Word);
var
 I: WORD;
 C: Char;
 S: String;
begin
 for I := 1 to Count do
   begin
     C:= ApdComPort1.GetChar;
     S:= S + C;
   end;
 Edit1.Text:= Edit1.Text + S;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 APdComPort1.Open:= False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 ApdComPort1.Open:= True;
end;

procedure TForm1.rtsClearButtonClick(Sender: TObject);
begin
 ApdComPort1.RTS:= False;
end;
```

```
procedure TForm1.rtsSetButtonClick(Sender: TObject);
begin
 ApdComPort1.RTS:= True;
end;
end.
```

Управление сигналом RTS выполняют обработчики нажатия кнопок `rtsSetButtonClick` и `rtsClearButtonClick`. Для установки сигнала RTS в высокий (активный) уровень свойство `RTS` компонента ApdComPort1 нужно установить в True, а для сброса сигнала на линии RTS этому же свойству следует присвоить значение False.

В терминальной программе, используемой для тестирования, должна быть активизирована опция разрешения аппаратного управления потоком данных.

В следующем, самом сложном примере, будет показано использование компонента TApdComPort в системном сервисе Windows. Системный сервис (назовем его SPortService) с интервалом в 10 сек будет передавать строку данных в последовательный порт COM4. Для тестирования приложения на порту COM1 должна быть запущена программа-эмулятор терминала, получающая отправленную строку.

Системные сервисы Windows — это особый класс приложений, которые могут выполняться в фоновом режиме при отсутствии пользователей в системе. Сервисы составляют большую часть процессов, работающих в операционной системе Windows, и предназначены для выполнения постоянных или периодических заданий. Прием или передача данных по коммуникационным каналам, в том числе и через последовательные порты — одна из распространенных задач, выполняемых системными сервисами.

Несмотря на некоторую сложность в разработке системных сервисов, в Delphi предусмотрен мастер создания такого класса приложений, который генерирует большую часть программного кода. Для создания системного сервиса в меню **File** следует выбрать опцию **New** и затем опцию **Other…**, после чего в окне **New Items** выбрать пиктограмму **Service Application** (рис. 4.14).

После создания шаблона системного сервиса поместим на форме приложения экземпляр ApdComPort1 компонента TApdComPort (рис. 4.15).

Для компонента ApdComPort1 установим стандартные свойства (скорость 9600 бод, без контроля четности, 8-битовый формат посылки с одним стоповым битом), а свойству `AutoOpen` присвоим значение False. Затем настроим параметры собственно сервиса (рис. 4.16).

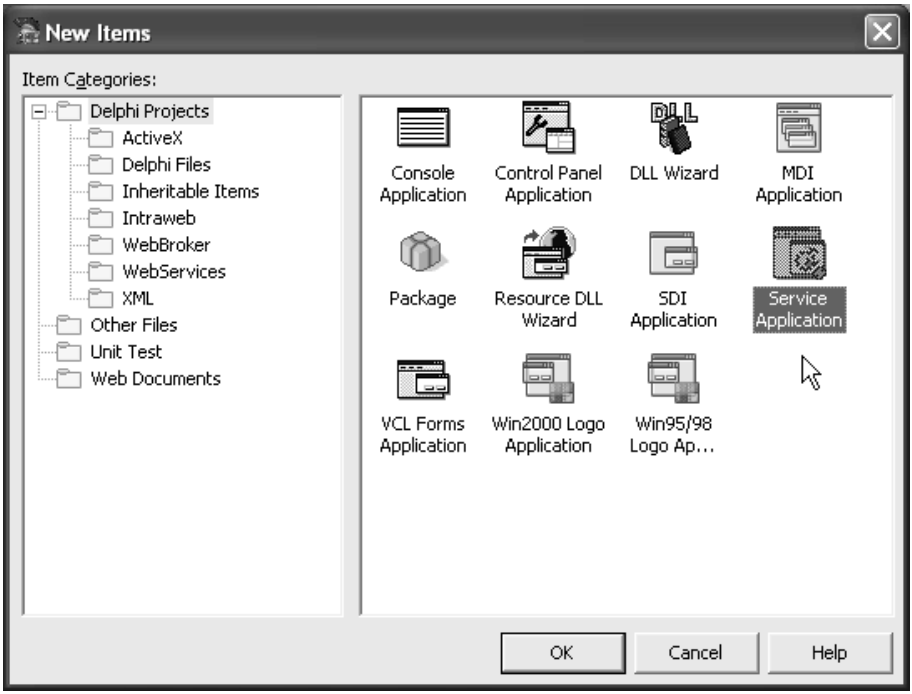


Рис. 4.14. Выбор приложения в мастере создания проектов

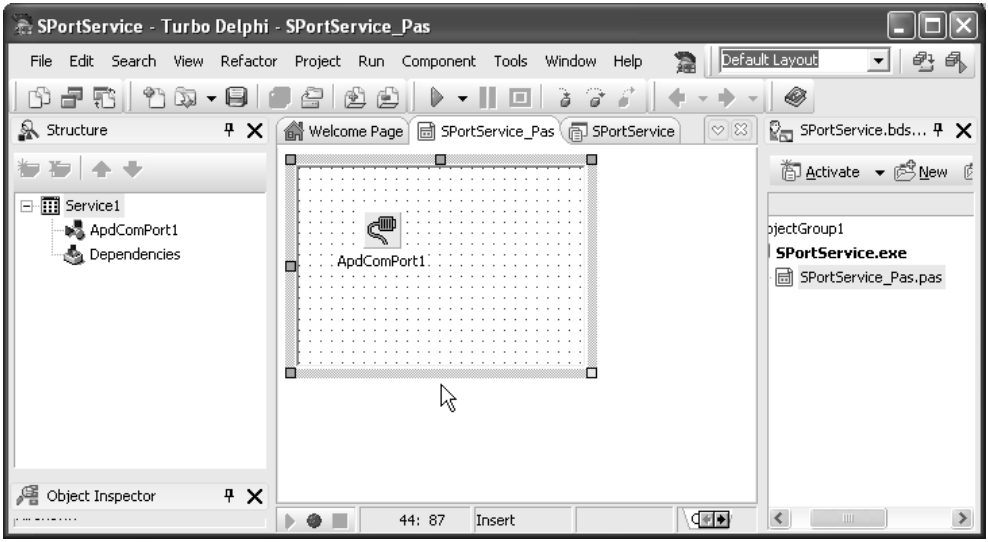


Рис. 4.15. Окно конструктора приложения

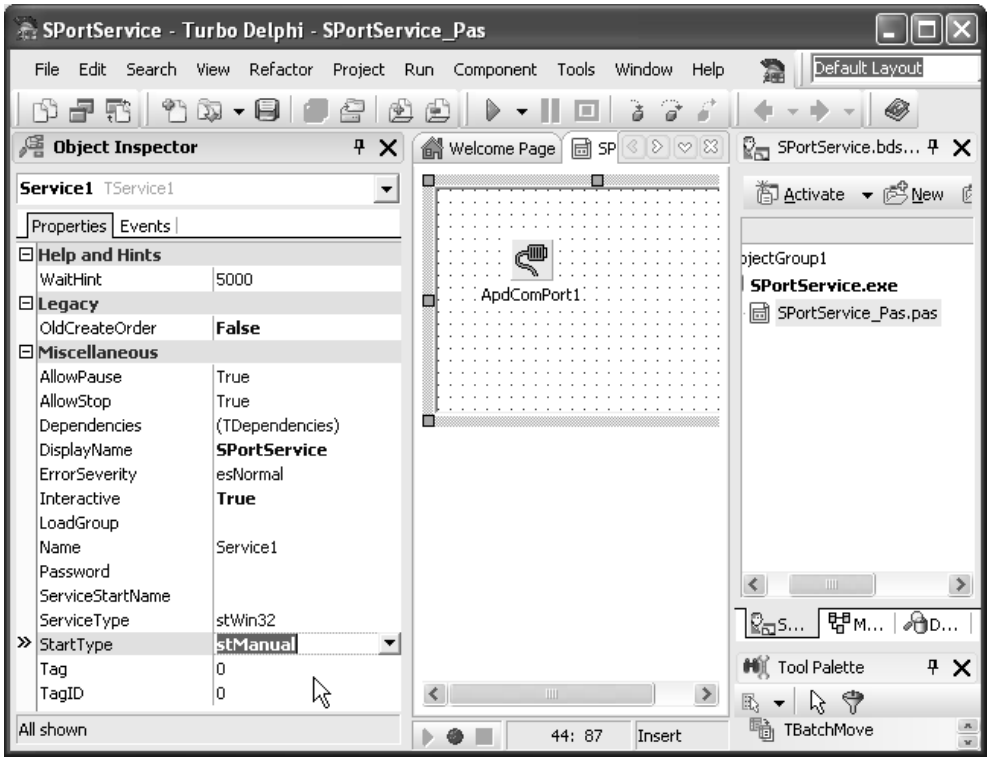


Рис. 4.16. Установка свойств системного сервиса Windows

Компонент Service1, инкапсулирующий функции системного сервиса, имеет множество свойств, но нам нужно присвоить определенные значения только отдельным, оставив остальным значения по умолчанию. Прежде всего, установим имя сервиса (свойство `DisplayName`), так как оно будет отображаться на системной консоли, доступной пользователю при выборе опции **Управление** после щелчка правой кнопкой мыши на пиктограмме **Мой компьютер**. В качестве имени выберем SPortService. Второе свойство, которое мы должны установить — тип запуска сервиса (**StartType**). Выберем ручной запуск, установив опцию **stManual**.

После установки свойств необходимо установить функции-обработчики событий сервиса. В простейшем случае достаточно определить обработчик события `OnExecute` (рис. 4.17).

Здесь функции-обработчику присваивается по умолчанию имя `ServiceExecute`. Программный код этой функции представлен в листинге 4.16.

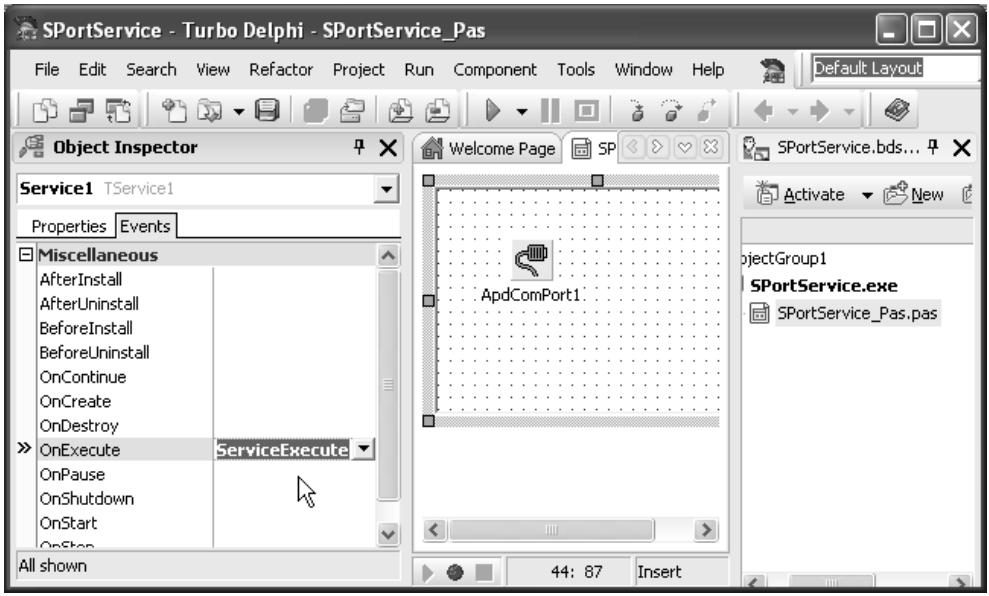


Рис. 4.17. Установка функции-обработчика сервиса

Листинг 4.16. Функция-обработчик события `OnExecute` системного сервиса

```
procedure TService1.ServiceExecute(Sender: TService);
begin
 ApdComPort1.Open:= True;
 while not Terminated do
 begin
   Service1.ServiceThread.ProcessRequests(False);
   Inc(Counter);
   ApdComPort1.Output:= 'SPortService is burned ' + IntToStr(Counter) + '
times'#13#10;
   Sleep(10000);
 end;
 ApdComPort1.Open:= False;
end;
```

Функция `ServiceExecute` запускается в основном потоке системного сервиса. Этого в нашем случае вполне достаточно, хотя в других случаях может пона-

добиться выполнять задания в специально создаваемом потоке. Собственно вся работа системного сервиса выполняется этой функцией. Обычно функция `ServiceExecute` выполняет задачу в цикле `while`, выход из которого происходит только при остановке сервиса. Функция каждые 10 сек передает в последовательный порт COM4 строку данных, для чего используется свойство `Output` компонента ApdComPort1. Необходимая задержка выполняется функцией `Sleep`. При запуске функции-обработчика необходимо открыть последовательный порт, что реализуется присвоением свойству `Open` компонента ApdComPort1 значения True. Перед завершением работы обработчика коммуникационный ресурс закрывается (значение False свойства `Open`).

После успешной компиляции и сборки проекта получим файл приложения SPortService.exe, который содержит программный код нашего сервиса. Для установки сервиса в системе нужно использовать ключ `/install`. В окне системной консоли нужно набрать строку

```
SPortService /install
```

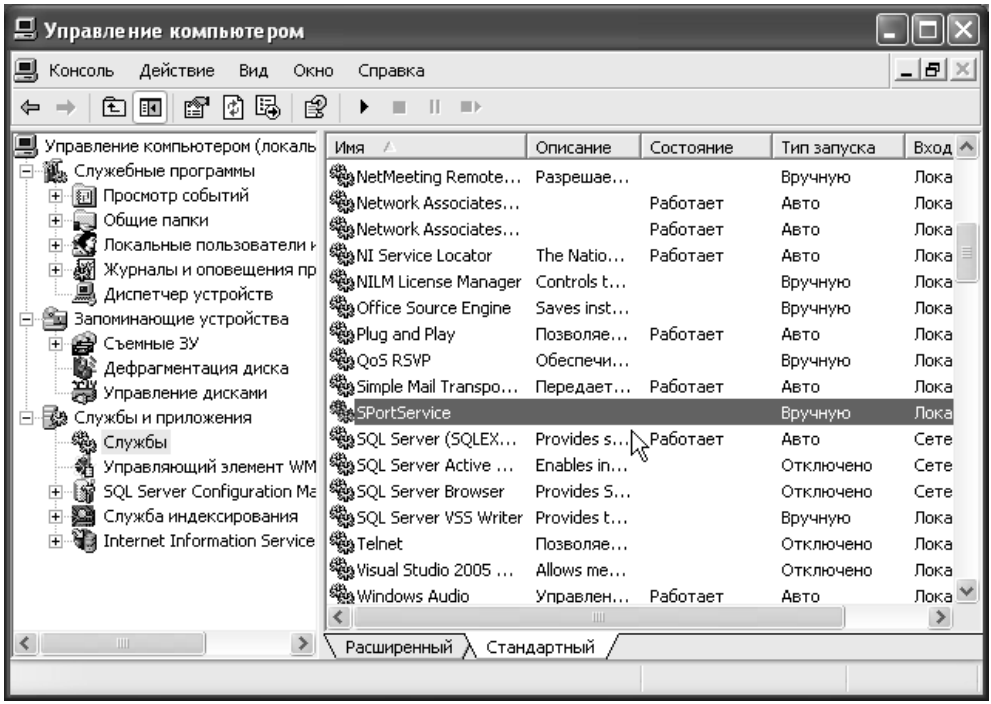


Рис. 4.18. Окно списка системных сервисов

При успешной установке сервиса появится соответствующее окно сообщения. Установленный сервис должен присутствовать в окне **Управление компьютером** при выборе опции **Управление** после щелчка правой кнопкой мыши на пиктограмме **Мой компьютер** (рис. 4.18).

Как видно из рисунка, сервис SPortService появился в списке системных сервисов. Для его запуска необходимо щелкнуть на выделенной строке правой кнопкой мыши и в выпадающем меню выбрать опцию **Пуск** (рис. 4.19).

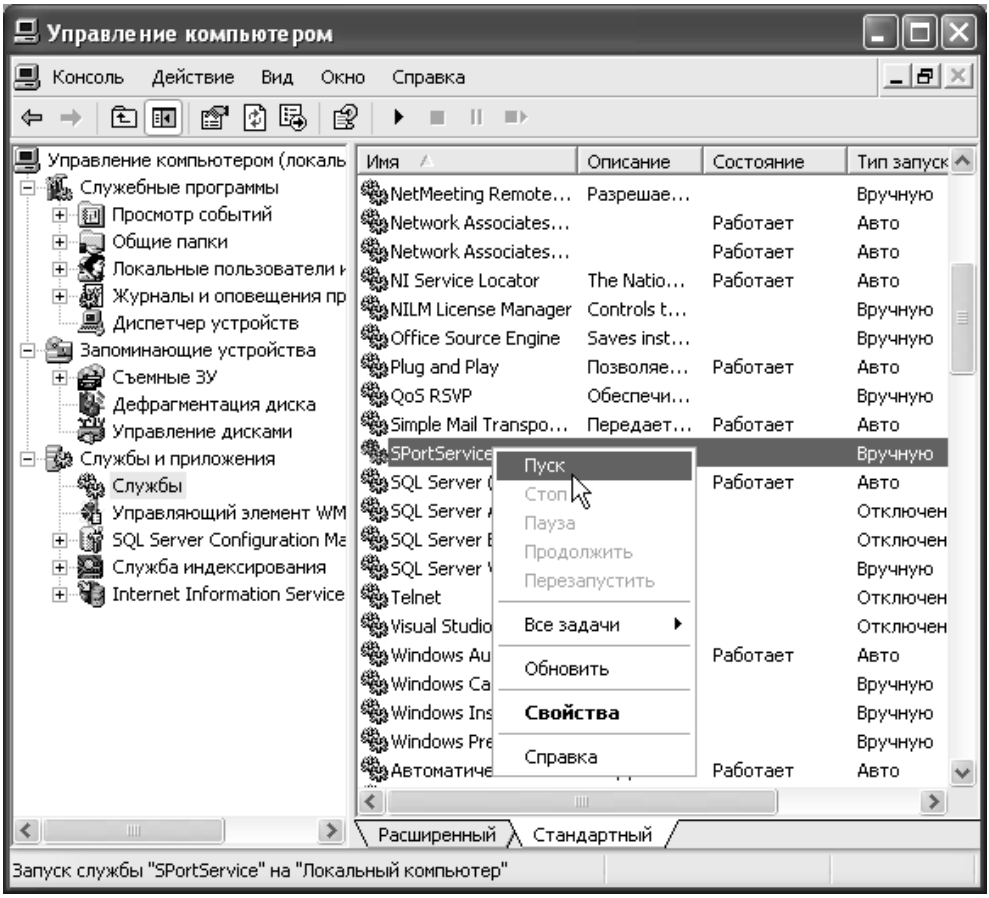


Рис. 4.19. Запуск системного сервиса SPortService

Для остановки системного сервиса необходимо выбрать строку с именем сервиса и после нажатия правой кнопки мыши в выпадающем меню выбрать

опцию **Стоп**. Для проверки работы нашего сервиса запустим терминальную программу, принимающую данные на порту COM1, и увидим результат (рис. 4.20).

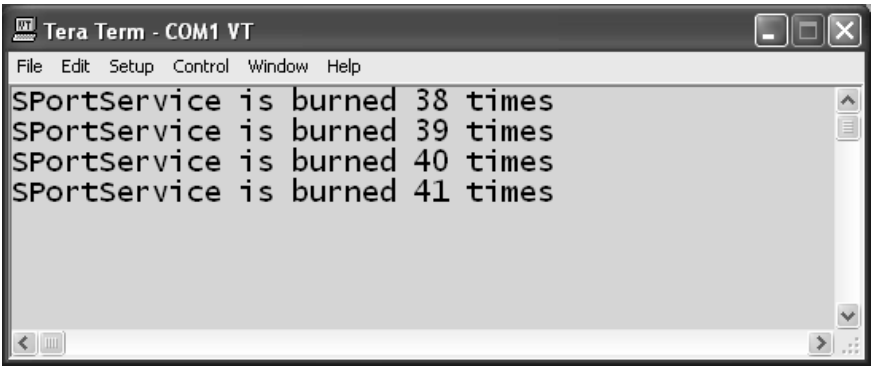


Рис. 4.20. Окно терминальной программы

Для деинсталляции системного сервиса предварительно нужно остановить его и выполнить команду

```
SPortService /uninstall
```

Компонент TApdComPort имеет много других полезных свойств, которые позволяют гибко управлять процессом обмена данными по последовательному порту. Дополнительную информацию по данному компоненту можно обнаружить в обширной документации на этот программный продукт.

4.3. Программирование последовательного ввода-вывода в среде Visual Studio .NET

Программирование обмена данными посредством последовательного интерфейса в приложениях с графическим интерфейсом, разработанных в среде Visual Studio .NET, существенно упростилось с появлением компонента SerialPort. Это дало возможность разработчикам программного обеспечения, создающим программы на языках C# и Visual Basic, тратить намного меньше усилий при написании приложений, работающих с последовательными портами ввода-вывода. Рассмотрим практические примеры разработки приложений с использованием этого компонента. Как и при разработке тестовых

примеров в Delphi, рабочая конфигурация аппаратно-программных средств будет такой:

- работающая с портом COM1 персонального компьютера программа-эмулятор терминала;
- работающее с портом COM4 расширителя интерфейса тестовое приложение. Если имеется свободный ПК с портом COM1, тестовое приложение можно запустить на нем, не забыв поменять в исходном тексте программы COM4 на COM1;
- нуль-модемное соединение с пятью линиями (сигналы RxD, TxD, RTS, CTS и общий провод).

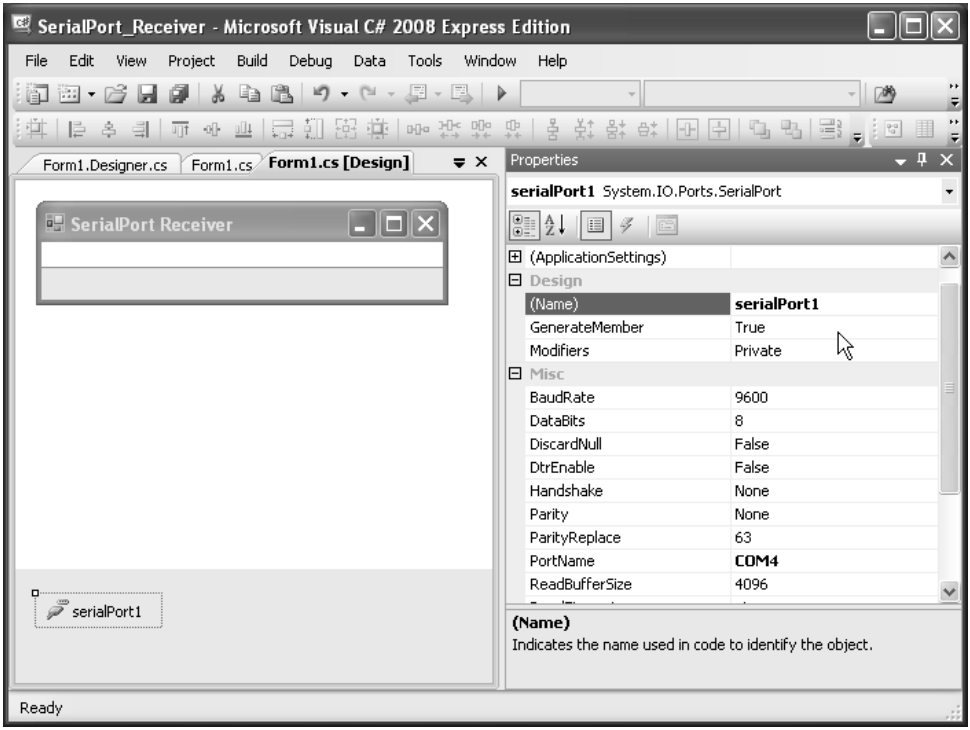


Рис. 4.21. Установка свойств компонента serialPort1

Наше первое приложение будет читать данные с последовательного порта COM4 и выводить их в окно текстового редактора. Разработка этого и последующих примеров выполняется на языке C#, а для разработки приложения

воспользуемся мастером проектов из среды программирования Microsoft Visual C# 2008 Express Edition, распространяемой бесплатно. Все проекты этого раздела можно компилировать и в более ранней версии среды разработки — Microsoft Visual Studio .NET 2005. На форму приложения перетащим из палитры компонентов **Toolbox** компоненты TextBox и SerialPort. Для экземпляра serialPort1 установим свойства так, как показано на рис. 4.21.

Почти все установки по умолчанию компонента serialPort1 нас устраивают, нужно изменить только имя порта PortName на COM4.

Перейдем к разработке исходного текста приложения. Для правильного использования компонента SerialPort при приеме данных с коммуникационного порта можно воспользоваться одной из двух методик:

- создать отдельный поток, при этом прием данных будет выполняться в функции потока;
- настроить обработчик события `DataReceived` компонента.

Следует учитывать, что корректная работа функции-обработчика события `DataReceived` возможна только при ее запуске в контексте дополнительного потока, особенно, если для отображения полученных данных используются графические компоненты, работа которых требует дополнительной синхронизации.

Мы рассмотрим оба варианта программирования и начнем с первого, а именно: создадим в приложении отдельный поток, функция-обработчик которого будет читать строку данных с порта COM4 и выводить ее содержимое в окно однострочного текстового редактора TextBox. Далее приводится исходный текст приложения (листинг 4.17).

Листинг 4.17. Программа приема строки данных с порта COM4 на языке C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;
```

```
namespace SerialPort_EXample1
{
  public partial class Form1 : Form
 {
  public String s1;
  public Thread myThread;
  public Form1()
   {
   InitializeComponent();
   }

  private void Form1_Load(object sender, EventArgs e)
   {
     serialPort1.Open();
     myThread = new Thread(this.ThreadProc);
     myThread.Start();
   }

   private void Form1_FormClosing(object sender,
                         FormClosingEventArgs e)
    {
     myThread.Abort();
     serialPort1.Close();
    }
   public void ThreadProc()
    {
      while (true)
       {
        s1 = serialPort1.ReadLine();
        textBox1.Text = s1;
       }
    }
  }
}
```

Поскольку в программе используется дополнительный поток, то в начале программы нужно указать ссылку на пространство имен `System.Threading`:

```
using System.Threading;
```

Новый поток `myThread` создается и запускается при открытии формы приложения (функция `Form1_Load`). Функция потока `ThreadProc` в бесконечном цикле ожидает поступления строки данных через последовательный порт и записывает полученные данные в переменную `s1` типа `String` с помощью функции

```
s1 = serialPort1.ReadLine();
```

Затем строка байтов отображается в окне редактора textBox1.

Кроме того, при запуске приложения необходимо открыть последовательный порт COM4 функцией `serialPort1.Open()`.

По завершении работы приложения поток уничтожается (функция `myThread.Abort()`), а последовательный порт закрывается (функция `serialPort1.Close()`).

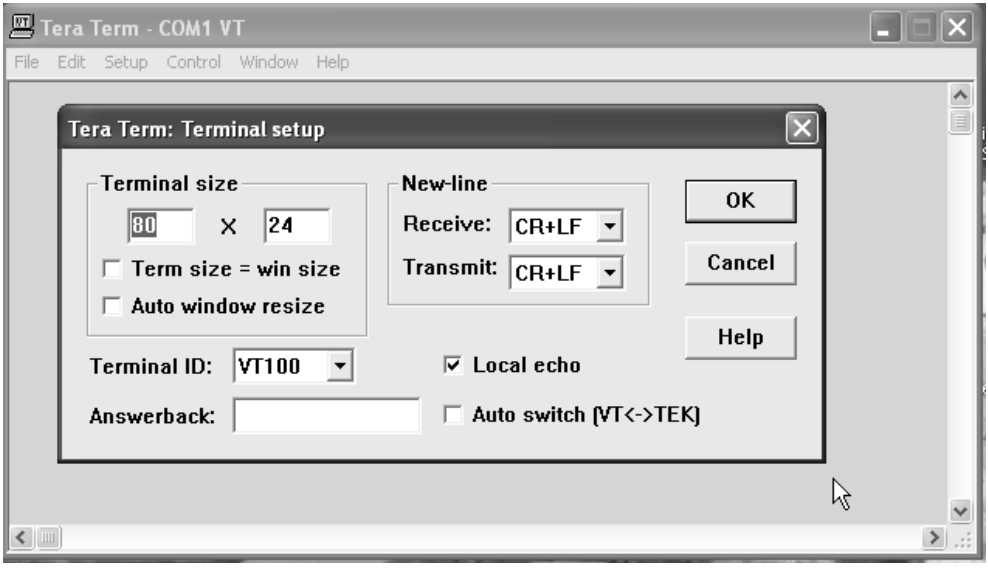


Рис. 4.22. Настройка терминальной программы

Для тестирования нашего приложения в программе-эмуляторе нужно установить признак конца строки как комбинацию символов возврата каретки

(CR — Carriade Return) и перевода строки (LF — Line Feed), что требуется для работы функции `ReadLine` компонента serialPort1. Установка символов конца строки в эмуляторе терминала Tera Term Pro, который используется в тестовых примерах, показана на рис. 4.22.

При использовании других программ-эмуляторов терминала подобные настройки, естественно, будут выполняться по-другому.

В следующем проекте демонстрируется обработка данных с последовательного порта при помощи обработчика события `DataReceived` компонента SerialPort. Вид окна приложения и используемые компоненты будут теми же, что и в предыдущем проекте, но исходный текст приложения будет другим (листинг 4.18).

Листинг 4.18. Прием данных с последовательного порта в обработчике `DataReceived`

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace SerialPort_EXample1
{
  public partial class Form1 : Form
   {
     public String s1;
     public Thread myThread;
     public Form1()
      {
       InitializeComponent();
      }
```

```
    private void Form1_Load(object sender, EventArgs e)
     {
       serialPort1.Open();
       myThread = new Thread(this.ThreadProc);
       myThread.Start();
     }

    private void serialPort1_DataReceived(object sender,
     System.IO.Ports.SerialDataReceivedEventArgs e)
     {
       s1 = serialPort1.ReadLine();
       textBox1.Text = "RECEIVED: " + s1;
     }

  private void Form1_FormClosing(object sender,
                        FormClosingEventArgs e)
   {
     myThread.Abort();
     serialPort1.Close();
   }
  public void ThreadProc()
   {
    this.serialPort1.DataReceived += new Sys-
tem.IO.Ports.SerialDataReceivedEventHandler
(this.serialPort1_DataReceived);
    while (true)
     {
     }
  }
 }
}
```

В этой программе для приема строки байтов запускается дополнительный поток с функцией `ThreadProc`. Но функция потока `ThreadProc` не выполняет непосредственный прием строки, а создает функцию-обработчик события

`DataReceived` с именем `serialPort1_DataReceived`, которая и выполняет необходимые действия с помощью оператора

```
this.serialPort1.DataReceived += new Sys-
tem.IO.Ports.SerialDataReceivedEventHandler
(this.serialPort1_DataReceived);
```

Таким образом, функция-обработчик `serialPort1_DataReceived` запускается в контексте дополнительного потока, что и требуется для корректной работы приложения.

Передача данных с помощью компонента SerialPort показана в следующем примере. Окно конструктора нашего приложения в этом случае будет выглядеть так, как показано на рис. 4.23.

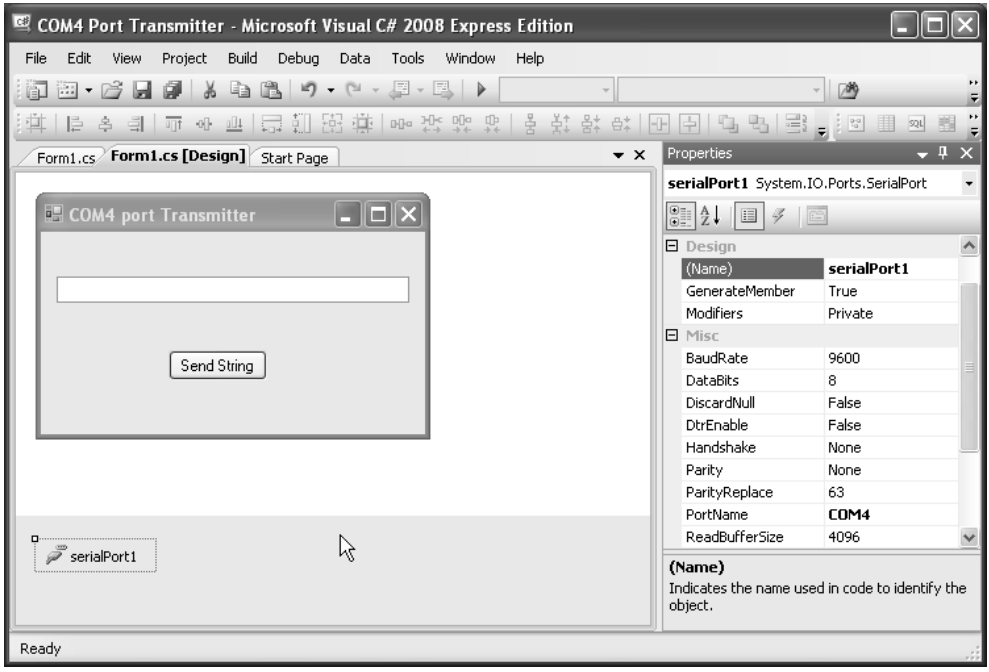


Рис. 4.23. Окно конструктора приложения

Текстовая строка вводится в окне однострочного редактора textBox1 и по нажатию кнопки **Send String** передается в последовательный порт COM4. Настройки компонента serialPort1 такие же, как и в предыдущих примерах. Исходный текст программы показан в листинге 4.19.

Листинг 4.19. Передача строки байтов в порт COM4 с помощью компонента SerialPort

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace COM4_Port_Transmitter
{
  public partial class Form1 : Form
   {
    public String s1;
    public Form1()
     {
      InitializeComponent();
     }

    private void Form1_Load(object sender, EventArgs e)
     {
      serialPort1.Open();
     }
    private void Form1_FormClosing(object sender,
                          FormClosingEventArgs e)
     {
      serialPort1.Close();
     }

    private void sendButton_Click(object sender, EventArgs e)
     {
      s1 = textBox1.Text;
```

```
        serialPort1.WriteLine(s1);
      }
   }
}
```

Исходный текст приложения несложен для понимания. Здесь присутствуют функции-обработчики `Form1_Load` (запуск приложения), `Form1_FormClosing` (завершение работы) и `sendButton_Click` (нажатие кнопки). В функции `Form1_Load` выполняется открытие последовательного порта, а функция `Form1_FormClosing` закрывает порт.

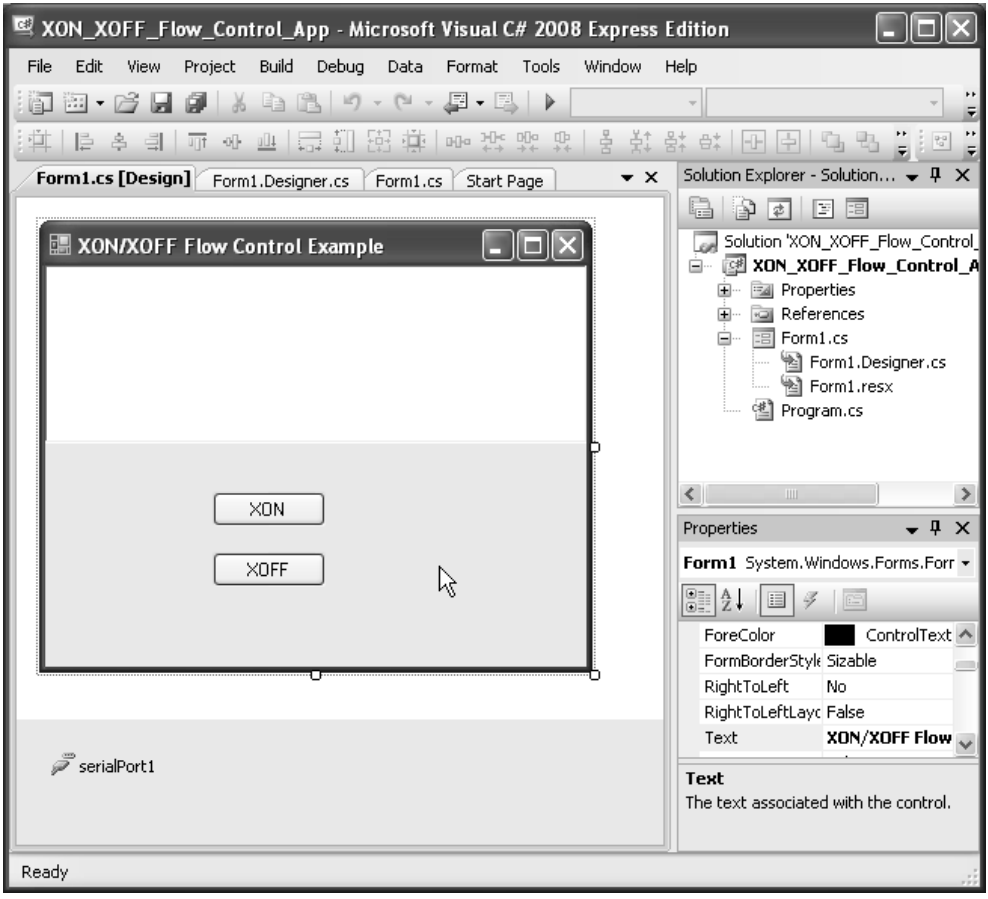


Рис. 4.24. Окно конструктора приложения

Передача строки выполняется в функции `sendButton_Click`. При этом содержимое окна редактирования textBox1 передается в последовательный порт функцией `serialPort1.WriteLine`.

С помощью установки свойств компонента SerialPort можно управлять потоком данных. Рассмотрим пример приложения, в котором реализован программный (XON/XOFF) метод управления потоком данных. Окно конструктора приложения показано на рис. 4.24.

Приложение работает следующим образом: при нажатии кнопки **XOFF** приложение посылает терминальной программе, работающей на порту COM1, байт XOFF, запрещая тем самым дальнейшую передачу байтов данных. При этом в окно текстового редактора выводится соответствующее сообщение. При нажатии кнопки **XON** разрешается дальнейшая передача данных приложению. Напомню, что для функционирования этой программы необходимо, чтобы опция, разрешающая XON/XOFF-управление потоком данных, была установлена в программе-эмуляторе. Исходный текст тестового приложения показан в листинге 4.20.

Листинг 4.20. Программное управление потоком данных с использованием компонента SerialPort

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace XON_XOFF_Flow_Control_App
{
  public partial class Form1 : Form
   {
    public byte [] buf = {17, 19};
    public Thread myThread;
    public String s1;
```

```
    public Form1()
     {
      InitializeComponent();
     }

    private void Form1_Load(object sender, EventArgs e)
     {
      serialPort1.Open();
      myThread = new Thread(this.ThreadProc);
      myThread.Start();
     }

    private void Form1_FormClosing(object sender,
                         FormClosingEventArgs e)
     {
      myThread.Abort();
      serialPort1.Close();
     }
    public void ThreadProc()
     {
      this.serialPort1.DataReceived += new Sys-
tem.IO.Ports.SerialDataReceivedEventHandler
(this.serialPort1_DataReceived);
      while (true) { }
    }

    private void xonButton_Click(object sender,
                                 EventArgs e)
     {
      serialPort1.Write(buf, 0, 1);
      richTextBox1.AppendText("Transmitting Enabled! ");
     }

    private void xoffButton_Click(object sender,
                                  EventArgs e)
      {
```

```
        serialPort1.Write(buf, 1, 1);
        richTextBox1.AppendText("Transmitting Disabled!  ");
       }

    private void serialPort1_DataReceived(object sender, Sys-
tem.IO.Ports.SerialDataReceivedEventArgs e)
     {
      s1 = serialPort1.ReadLine();
      richTextBox1.AppendText(s1);
     }
  }
}
```

Часть исходного текста программы читателю уже знакома по предыдущим примерам, поэтому остановлюсь только на сделанных модификациях. В программе объявлен массив `buf`, состоящий из двух элементов. Нулевой элемент массива имеет значение 17, что соответствует стандартному значению XON, а первый элемент — значение 19 (XOFF). Приложение принимает байты данных через последовательный порт COM4 и отображает их в виде отдельных строк в текстовом редакторе RichTextEdit. Для приема данных от терминальной программы, работающей с портом COM1, создается отдельный поток, в контексте которого запускается обработчик события `DataReceived`.

Байты XON или XOFF отправляются передатчику при нажатии кнопок **XON** или **XOFF** на форме приложения, при этом вызывается функция-обработчик `xonButton_Click` или `xoffButton_Click`. Для отправки байта XON служит функция

```
serialPort1.Write(buf, 0, 1);
```

а байт XOFF передается терминальной программе с помощью функции

```
serialPort1.Write(buf, 1, 1);
```

При передаче байтов XON/XOFF в окно текстового редактора RichTextBox выводится соответствующее сообщение.

Для выполнения аппаратного (RTS/CTS) управления потоком данных программа, получающая данные, разрешает передачу данных, устанавливая сигнал на линии RTS в активное состояние (обычно это соответствует высокому уровню напряжения), или запрещает их передачу установкой низкого уровня напряжения.

Следующее приложение запрещает или разрешает передачу данных программой-эмулятором терминала, устанавливая или сбрасывая сигнал на линии RTS. Окно конструктора приложения показано на рис. 4.25.

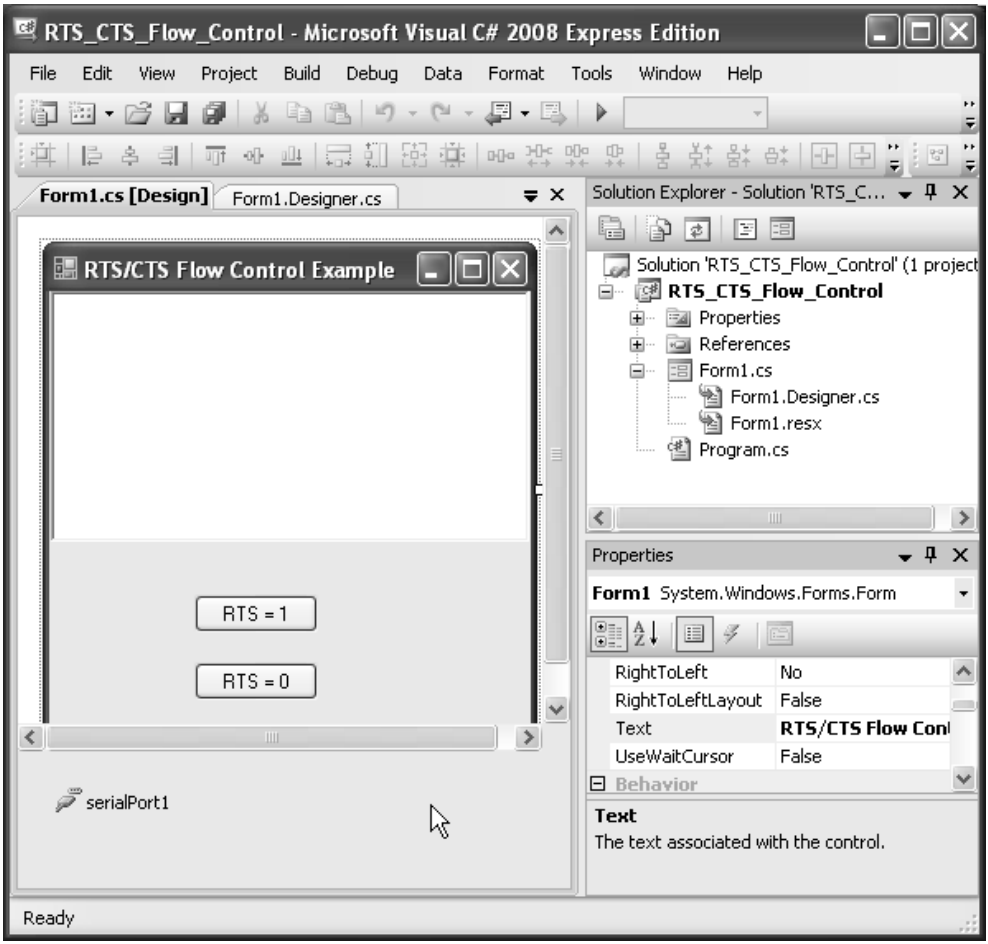


Рис. 4.25. Окно конструктора приложения

В приложении используются те же компоненты, что и в предыдущем примере. При нажатии кнопки **RTS = 1** сигнал на линии RTS устанавливается в активное состояние, а при нажатии кнопки **RTS = 0** сигнал на этой линии становится неактивным.

Программный код приложения во многом напоминает тот, что использовался в предыдущем примере, за исключением кода обработчиков нажатия кнопок. Исходный текст обработчика нажатия кнопки RTS = 1 показан далее:

```
private void rtsOnButton_Click(object sender,
                               EventArgs e)
  {
   serialPort1.RtsEnable = true;
   richTextBox1.AppendText("Data Receiving Enabled! ");
  }
```

Здесь установка сигнала на линии RTS в активное состояние выполняется посредством установки свойства `RtsEnable` компонента serialPort1 в True. Обработчик нажатия кнопки **RTS = 0** имеет следующий исходный текст:

```
private void rtsOffButton_Click(object sender, EventArgs e)
  {
   serialPort1.RtsEnable = false;
   richTextBox1.AppendText("Data Receiving Disabled! ");
  }
```

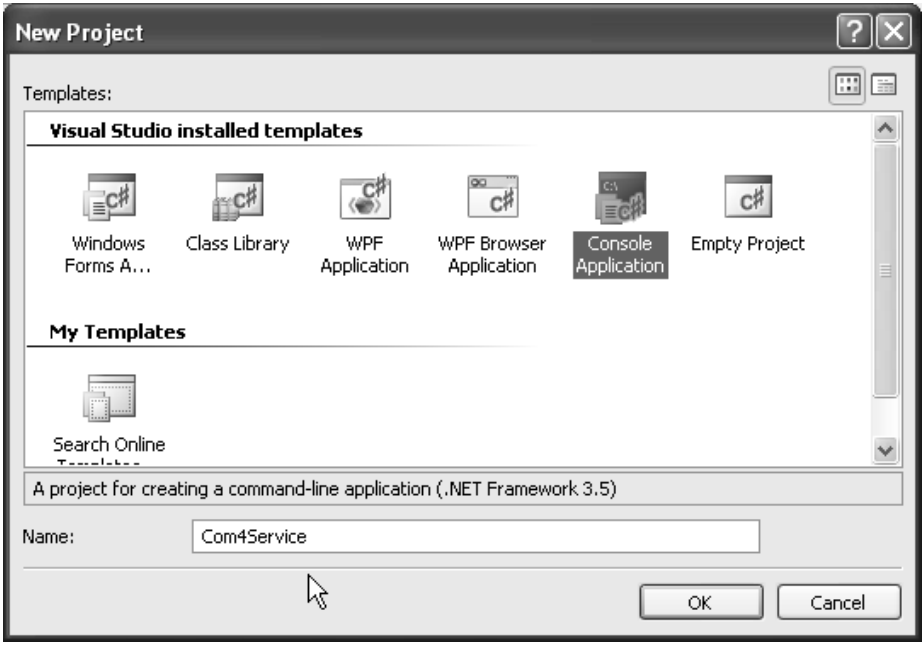


Рис. 4.26. Выбор типа приложения

Наш следующий проект с использованием компонента SerialPort намного более сложный по сравнению с предыдущими и демонстрирует прием данных с коммуникационного порта COM4 и запись полученной строки в системный файл журнала событий "Приложение" (Application). Содержимое этого файла можно затем просмотреть, открыв окно системной консоли управления **Управление компьютером**. Для записи принятой с порта COM4 информации в файл журнала событий разработаем системный сервис Windows, который назовем Com4Service. Системные сервисы создаются как консольные приложения, поэтому в качестве шаблона выберем консольное приложение (рис. 4.26).

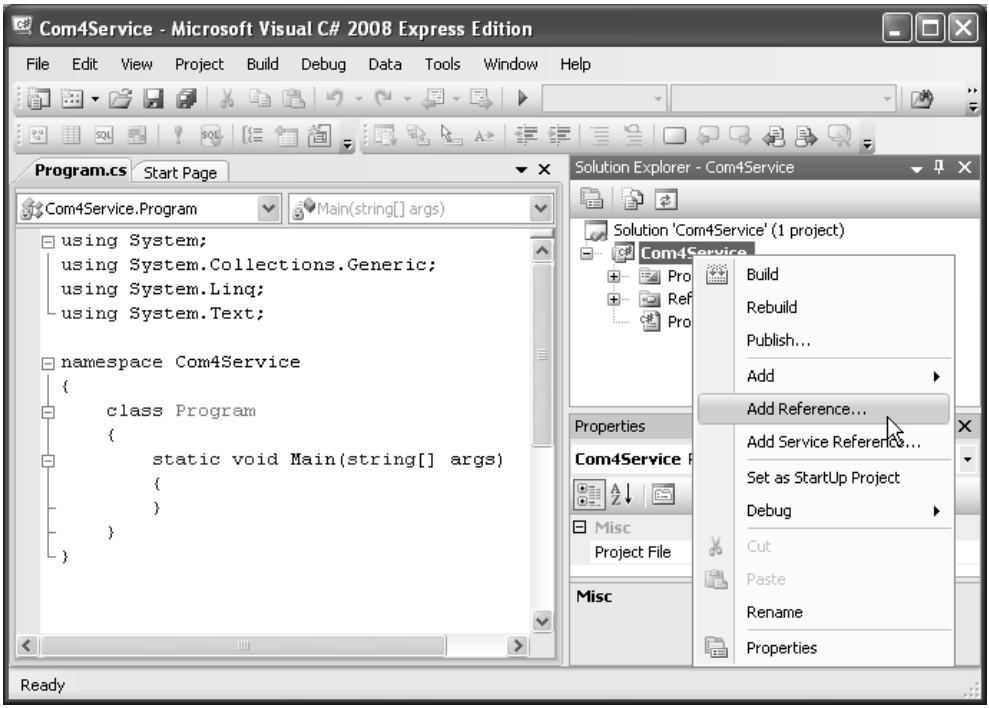


Рис. 4.27. Добавление ссылки на пространство имен

После создания шаблона проекта файл Program.cs будет содержать следующий программный код:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

namespace Com4Service
{
  class Program
  {
    static void Main(string[] args)
     {
     }
  }
}
```

Поскольку мы разрабатываем системный сервис, то в проект необходимо включить ссылку на пространство имен `System.ServiceProcess`, которое содержит классы для разработки данного типа приложений. Добавление ссылки показано на рис. 4.27 и 4.28.

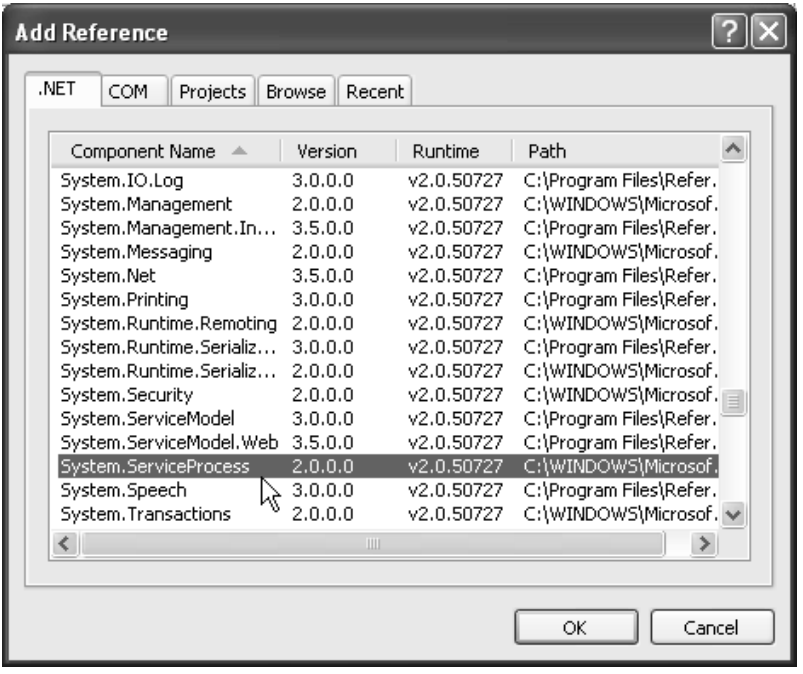


Рис. 4.28. Выбор пространства имен `System.ServiceProcess`

Для функционирования консольного приложения как системного сервиса необходимо включить в проект экземпляр класса `ServiceBase`. Вначале просто включим новый класс в наш проект (рис. 4.29 и 4.30).

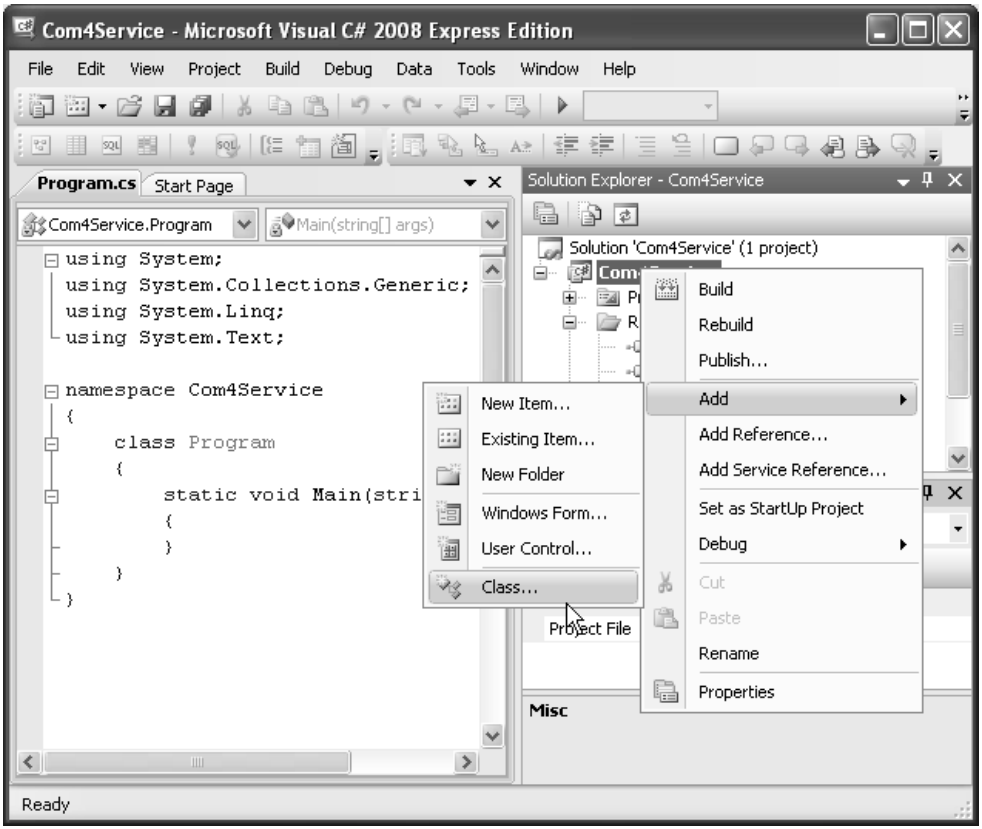


Рис. 4.29. Включение класса в проект

Мастер включения шаблонов одновременно с выбором пустого класса помещает код конструктора в файл Class1.cs. Примем предлагаемое имя файла с исходным текстом и нажмем кнопку **Add**. Система создаст файл Class1.cs со следующим исходным текстом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace Com4Service
{
    class Class1
    {
    }
}
```

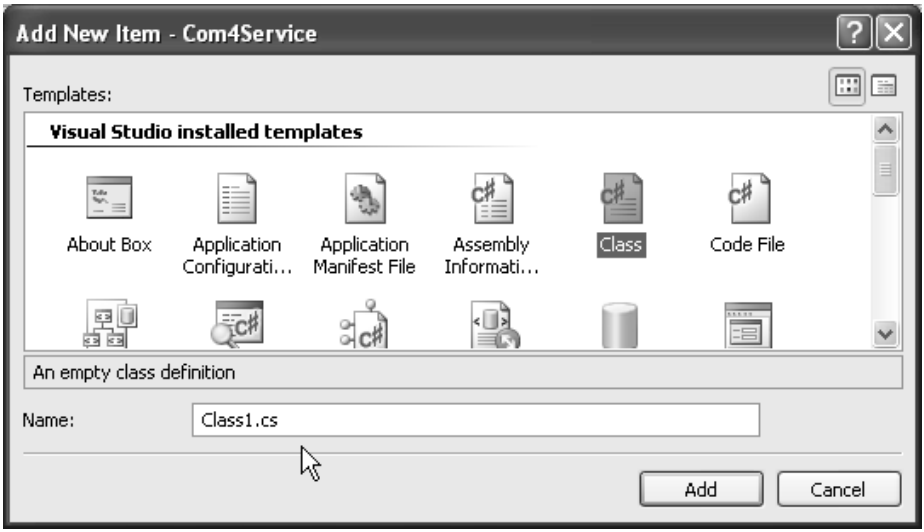


Рис. 4.30. Выбор класса

Все наши дальнейшие действия будут связаны с модификацией свойств и методов класса `Class1`. Класс `Class1` должен наследовать `ServiceBase`, и, кроме того, нужно импортировать пространства имен `ServiceProcess`, `Threading`, `IO.Port` и `Diagnostics`. После внесенных изменений исходный текст будет выглядеть так:

```
using System;
using System.ServiceProcess;
using System.Threading;
using System.IO.Ports;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace Com4Service
{
    class Class1:ServiceBase
    {
    }
}
```

В окне **Solution Explorer** проекта выберем опцию **View Designer** и перейдем в окно конструктора. Здесь в окне **Properties** в поле **ServiceName** введем имя системного сервиса, которое в данном случае будет **Com4Service** (рис. 4.31).

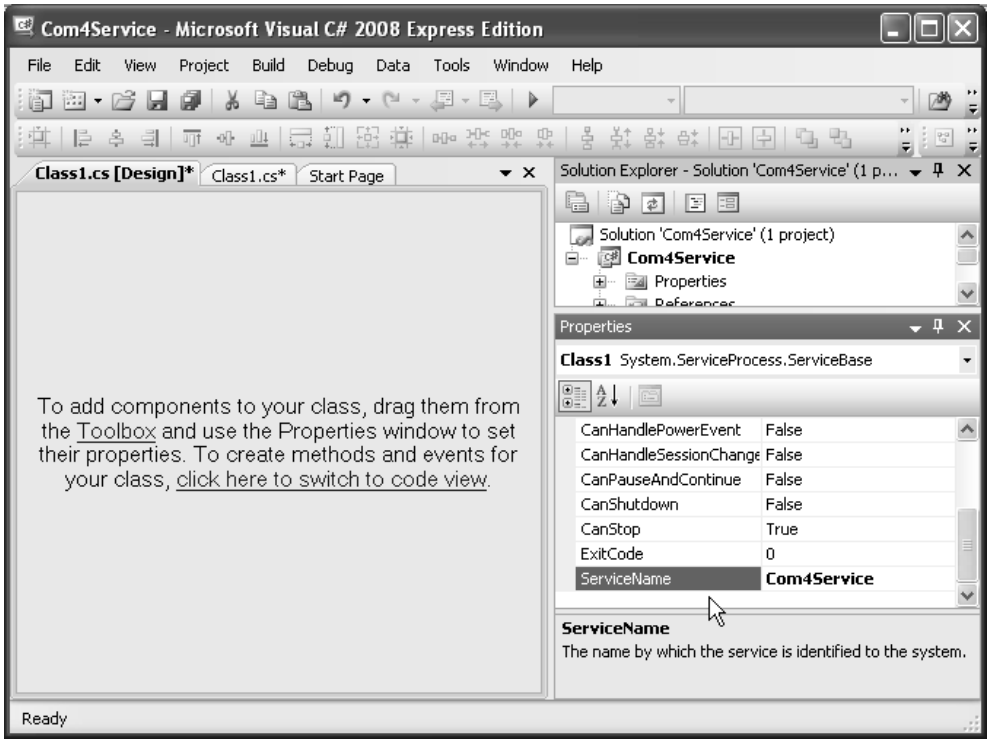


Рис. 4.31. Установка параметра `ServiceName`

Далее переопределим методы `OnStart` и `OnStop` базового класса `ServiceBase` в классе `Class1` и добавим программный код для обработки данных с последовательного порта. Полный исходный текст, в котором определены необходимые методы класса `Class1`, приведен в листинге 4.21.

Листинг 4.21. Программный код класса `Class1`

```
using System;
using System.ServiceProcess;
using System.Threading;
using System.IO.Ports;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Com4Service
{
   class Class1:ServiceBase
   {
     public static Thread myThread;
     public static SerialPort serialPort1;
     public static EventLog ev;

     private void InitializeComponent()
      {
        // Class1
        //
        this.ServiceName = "Com4Service";
      }
     protected override void OnStart(string[] args)
      {
        base.OnStart(args);
        serialPort1 = new SerialPort("COM4", 9600,
                                     Parity.None, 8,
                                     StopBits.One);
        serialPort1.Open();
        ev = new EventLog("Application",
                  ".", "Com4Service");
        myThread = new Thread(ThreadProc);
        myThread.Start();
      }
```

```
    protected override void OnStop()
     {
      base.OnStop();
      serialPort1.Close();
     }
    public static void ThreadProc()
     {
       serialPort1.DataReceived +=new
        SerialDataReceivedEventHandler
        (serialPort1_DataReceived);
       while (true);
   }

    public static void serialPort1_DataReceived(object sender, SerialDa-
taReceivedEventArgs e)
     {
      String s = serialPort1.ReadLine();
      ev.WriteEntry("RECEIVED: " + s);
     }
  }
}
```

Наш сервис должен записывать строку данных, приходящую с последовательного порта COM4, в файл журнала Application (Приложение), поэтому в программный код класса `Class1` необходимо включить экземпляр класса `EventLog` с именем `ev`. С помощью метода (функции) `WriteEntry` этого класса строка будет записываться в файл журнала. Все операции с последовательным портом выполняет класс `SerialPort` с помощью функции-обработчика `serialPort1` события `DataReceived`. Сам обработчик инициализируется в функции потока `ThreadProc`.

Основные функции инициализации, включая открытие последовательного порта COM4, выполняются в обработчике события `OnStart` системного сервиса Com4Service. При остановке сервиса (событие `OnStop`) последовательный порт закрывается.

После создания программного кода класса `Class1` необходимо запустить сервисное приложение, что должно выполняться в функции `Main` основной программы.

Модифицированный вариант исходного текста основной программы приведен далее:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Com4Service
{
  class Program
   {
    static void Main(string[] args)
     {
       System.ServiceProcess.ServiceBase.Run
        (new Com4Service.Class1());
     }

}
```

Компиляция и сборка приложения выполняются обычным образом, а для инсталляции сервиса в операционной системе Windows можно воспользоваться стандартной системной утилитой sc.exe. Предположим, что наше приложение записано в файл com4service.exe на диск C. Тогда для инсталляции системного сервиса нужно выполнить командную строку

```
sc create com4service biPath= c:\com4service.exe
```

Системный сервис можно запустить либо из консоли управления, выбрав опцию **Пуск**, либо набрав команду

```
net start com4service
```

Остановить сервис можно либо из консоли управления Windows, либо набрав команду

```
net stop com4service
```

Для деинсталляции системного сервиса нужно использовать утилиту sc.exe:

```
sc delete com4service
```

Для тестирования приложения надо запустить программу-эмулятор терминала и передать какую-либо строку через порт COM1 (COM1 и COM4 соедине-

ны вместе). Посмотреть на результат работы системного сервиса можно из консоли управления, просмотрев записи журнала Application (рис. 4.32).

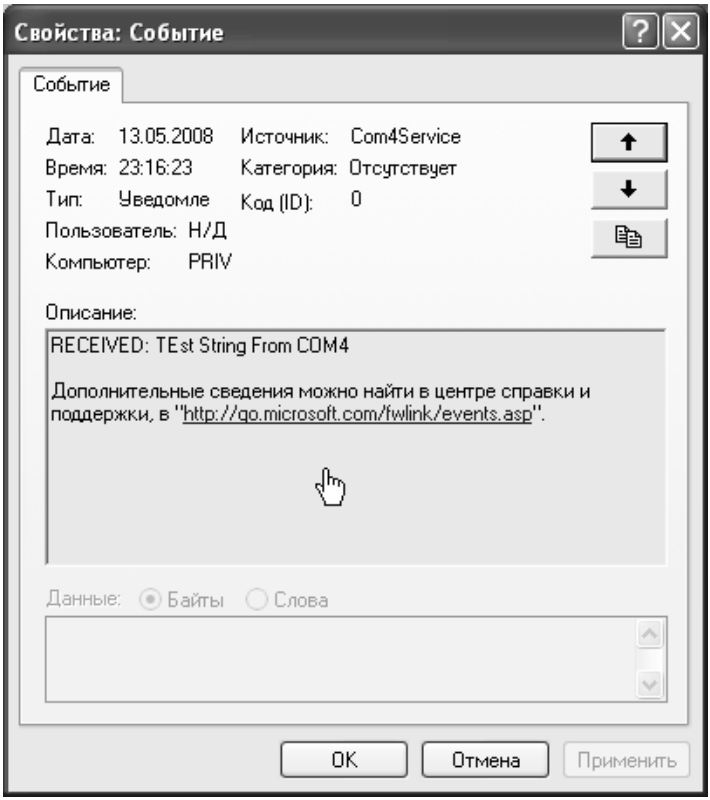


Рис. 4.32. Запись в журнале событий

Предыдущий проект можно расширить с точки зрения практической ценности, если записывать данные, полученные из последовательного порта, в текстовый файл. При этом в файл журнала приложений можно записывать строку уведомления о получении данных. Наш модифицированный системный сервис назовем Com4Service2 и при его создании воспользуемся базовым файлом исходного текста Class1.cs из предыдущего проекта, включив необходимый программный код в класс `Class1`.

Модифицированный исходный текст файла Class1.cs будет выглядеть так, как показано в листинге 4.22.

Листинг 4.22. Модифицированный исходный текст файла Class1.cs

```
using System;
using System.ServiceProcess;
using System.Diagnostics;
using System.IO;
using System.IO.Ports;
using System.Threading;

using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Com4Service2
{
  class Class1: ServiceBase
  {
   public static Thread myThread;
   public static EventLog ev;
   public static StreamWriter sw;
   public static SerialPort serialPort1;

   private void InitializeComponent()
   {
     //
     // Class1
     //
    this.ServiceName = "Com4Service2";
   }
   protected override void OnStart(string[] args)
   {
    base.OnStart(args);
    ev = new EventLog("Application", ".", "Com4Service2");
    serialPort1 = new SerialPort("COM4", 9600,
               Parity.None, 8, StopBits.One);
    serialPort1.Open();
```

```
    sw = new StreamWriter("c:\\SWLOG.txt", true);
    sw.AutoFlush = true;
    myThread = new Thread(ThreadProc);
    myThread.Start();
   }
   protected override void OnStop()
    {
     base.OnStop();
     sw.Close();
     serialPort1.Close();
    }

   public static void ThreadProc()
   {
    serialPort1.DataReceived +=new
             SerialDataReceivedEventHandler
             (serialPort1_DataReceived);
    while (true);
   }
   public static void serialPort1_DataReceived
                  (object sender,
                  SerialDataReceivedEventArgs e)
   {
    String s1 = serialPort1.ReadLine();
    sw.WriteLine(s1);
    ev.WriteEntry("DATA RECEIVED From COM4.");
   }
 }
}
```

Большая часть исходного текста нам знакома по предыдущему примеру, поэтому рассмотрим только внесенные изменения. Для создания текстового файла и записи в него данных используется объект `StreamWriter`. С помощью оператора

```
sw = new StreamWriter("c:\\SWLOG.txt", true);
```

в обработчике события `OnStart` системного сервиса Com4Service2 на диске C создается новый файл с именем SWLOG.txt, в который и будут записываться данные с последовательного порта COM4. Второй параметр, равный True, указывает на возможность добавления данных в файл, если таковой уже существует.

Запись полученных данных в файл осуществляется в функции-обработчике `serialPor1_DataReceived` события `DataReceived`. В этой же функции в файл журнала приложений записывается уведомляющее сообщение.

При остановке системного сервиса (событие `OnStop`) наряду с другими необходимыми действиями следует закрыть файл, что выполняет оператор

```
sw.Close();
```

Свойство `AutoFlush` объекта `StreamWriter` дает возможность просматривать содержимое файла SWLOG.txt при работающем сервисе Com4Service2.

Инсталляция, деинсталляция, запуск и тестирование системного сервиса Com4Service2 выполняется по той же методике, что была использована для Com4Service из предыдущего примера.

4.4. Последовательный обмен данными и сети TCP/IP

При построении разветвленных промышленных и лабораторных сетей сбора и обработки информации возникает необходимость в передаче данных удаленным узлам, географически расположенным в разных точках. Например, довольно часто возникает необходимость анализа данных, полученных с промышленного или лабораторного модуля, подсоединенного по интерфейсу RS-232 или RS-485 к персональному компьютеру, на другом компьютере, расположенном в другом районе города или в другой местности. Проблема легко решается, если компьютеры соединены в сеть. Как правило, стандартным решением является использование сети Интернет или, по-другому, сети на базе стека протоколов TCP/IP. Схема реализации простейшей системы сбора и передачи данных, в которой задействованы как минимум два компьютера и внешнее устройство, показана на рис. 4.33.

По этой схеме данные передаются по интерфейсу RS-232 или RS-485 от внешнего устройства, в качестве которого может выступать и персональный компьютер, на ПК с запущенным сетевым приложением (сетевой клиент). Клиент передает данные по сети на другой компьютер, на котором

работает сетевая программа-сервер. Для взаимодействия между программой-клиентом и программой-сервером используется протокол TCP/IP. В дальнейшем программу-клиент будем называть "клиент TCP", а программу-сервер — "сервер TCP".

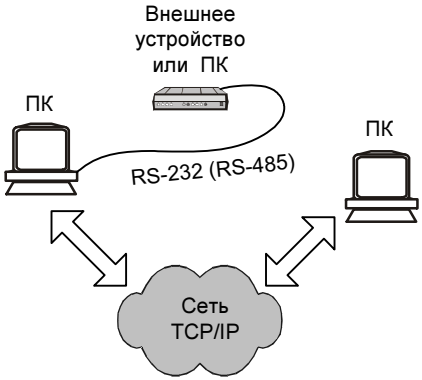


Рис. 4.33. Схема передачи данных по сети TCP/IP

Клиент TCP в такой конфигурации выполняет две функции:

- обрабатывает данные, которые передаются к внешнему устройству по интерфейсу RS-232 или принимаются от него;
- передает обработанные данные по сети серверу TCP или же принимает от него команды.

Направление передачи и обработки данных в такой конфигурации может быть любым. Во многих случаях клиент TCP выполняет самостоятельное управление внешним устройством и передает обработанные данные серверу с определенной периодичностью. В целом ряде случаев клиент TCP управляет внешним устройством по командам удаленного сервера. Нередко выбирают конфигурацию, в которой внешнее устройство подсоединено к серверу TCP, который собирает данные и передает их по запросу TCP-клиента.

Не вдаваясь в подробности функционирования протокола TCP/IP, поскольку это сложная тема и требует серьезного изучения, вкратце объясню принцип взаимодействия сетевого клиента и сетевого сервера. Для этого обратимся к рис. 4.34.

Взаимодействие между любыми сетевыми приложениями осуществляется посредством специальных объектов, которые называются *сокетами*. Сокет

("гнездо") служит "адресом", по которому каждое сетевое приложение идентифицирует себя в сети. Сокет включает IP-адрес (интернет-адрес) хоста, на котором работает приложение, и номер порта. Сервер TCP осуществляет "прослушивание" запросов на соединение от клиентов на специально выделенном сокете. При поступлении запроса сервер создает рабочее соединение с клиентом, по которому осуществляется последующий обмен данными, и продолжает прослушивать входящие запросы (см. рис. 4.34). При этом для клиентского соединения создается новый сокет с тем же IP-адресом, но с другим номером порта.

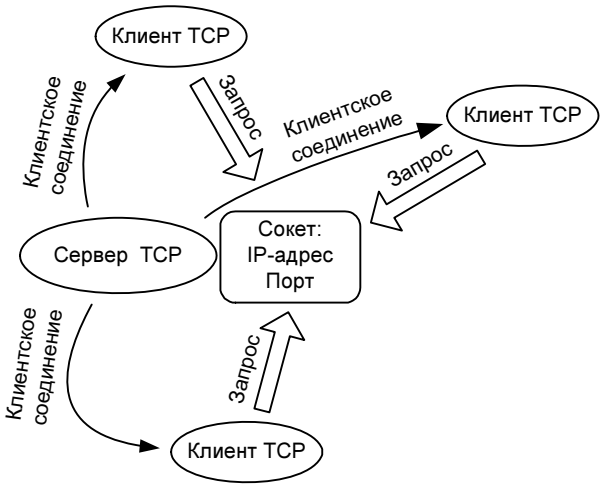


Рис. 4.34. Схема взаимодействия сервера и клиентов TCP

Предположим, например, что сервер сетевой службы Telnet прослушивает входящие запросы на хосте с IP-адресом 194.15.27.12, используя для этого стандартный порт 23. При поступлении запроса от сетевого клиента сервер Telnet может выделить клиенту порт 4997, по которому будет осуществляться обмен данными. Следующий клиент получит другой свободный порт, например, 4999. Таким образом, первый клиент будет работать на сокете 194.15.27.12:4997, а второй — на сокете 194.15.27.12:4999. Сервер Telnet будет продолжать принимать запросы на порту 23, создавая новые соединения. По завершению сеанса работы с клиентом рабочий сокет должен быть закрыт.

Разработка сетевых приложений — сложная задача, но реализовать ее можно довольно просто, если использовать современные инструментальные средст-

ва, такие как Visual Studio .NET и Delphi. Обе среды программирования предоставляют разработчику классы и компоненты, существенно облегчающие создание сетевых приложений, поскольку не требуется знать деталей сетевого взаимодействия, как в случае использования специальных библиотек сетевых функций.

Разработаем в Visual C# проект, состоящий из двух сетевых приложений, одно из которых выступает в роли сервера TCP, получающего данные по сети, а другое — в роли клиента TCP, собирающего данные с последовательного порта COM4. Клиент TCP получает строку байтов с последовательного порта, создает сетевое подключение к серверу, прослушивающему запросы на TCP-порту 5151 (это значение выбрано произвольным из свободных номеров), и отправляет данные серверу. Сервер TCP "прослушивает" запросы на подключение и при поступлении такого запроса создает новое клиентское соединение, через которое принимает данные от клиента. Полученные сервером данные отображаются на экране консоли.

Несколько слов о тестировании программного обеспечения нашего проекта: его можно выполнить на одной машине, располагающей двумя свободными COM-портами, запустив приложение-сервер и приложение-клиент. Автор использует плату расширения NetMos 9835 PCI Multi-I/O Controller. Если имеется два компьютера, подключенных к локальной сети, то на одном из них можно запустить программу-сервер, а на другом — клиентское приложение.

Рассмотрим программный код приложения-сервера TCP. Исходный текст сервера TCP показан в листинге 4.23.

Листинг 4.23. Программный код приложения-сервера TCP

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TCPServer
{
  class Program
```

```
  {
   static void Main(string[] args)
   {
    int bytesRead = 0;
    byte[] buf = new byte[1024];
    String s1;

    IPAddress addr = IPAddress.Parse("127.0.0.1");
    IPEndPoint iep = new IPEndPoint(addr, 5151);
    TcpListener server = new TcpListener(iep);
    server.Start();
    Console.WriteLine("Waiting for data on port 5151...");
    while (true)
     {
      bytesRead = 0;
      TcpClient client = server.AcceptTcpClient();
      NetworkStream stream = client.GetStream();
      bytesRead = stream.Read(buf, 0, buf.Length);
      if (bytesRead > 0)
       {
       s1 = System.Text.ASCIIEncoding.ASCII.GetString
           (buf, 0, bytesRead);
       Console.WriteLine("DATA RECEIVED: " + s1);
       }
      stream.Close();
      client.Close();
     }
     server.Stop();
    }
  }
}
```

Для программирования серверного приложения TCP импортируем в наше приложение пространство имен `System.Net`. Простейшее сетевое серверное приложение работает по стандартной схеме: вначале создается прослушивающий сокет, который принимает запросы сетевых клиентов на подключе-

ние, после чего для каждого отдельного клиента создается "рабочий" сокет, через который и выполняется обмен данными с TCP-сервером. Прослушивающий сокет "сидит" в бесконечном цикле и не участвует в обмене данными, его функция — отследить поступающий запрос и инициировать создание нового подключения.

В нашей программе-сервере прослушивающий сокет с именем `server` создается оператором

```
TcpListener server = new TcpListener(iep);
```

Сокет привязывается к конкретному хосту с помощью IP-адреса и номера порта, на котором выполняется прослушивание. Оба эти параметра содержатся в структуре `iep`. В качестве IP-адреса в серверном и клиентском приложениях выбран адрес 127.0.0.1 интерфейса "обратной связи" (loopback interface), который инициируется операционной системой для диагностических целей. После выполнения этого оператора нужно разрешить работу сокета:

```
server.Start();
```

Сокет работает в блокирующем режиме и ожидает запросов в цикле `while`. При подключении нового клиента инициируется новое соединение:

```
TcpClient client = server.AcceptTcpClient();
```

Посредством сокета `client` будет выполняться чтение данных, поступающих от клиента, для чего создается новый поток `stream`:

```
NetworkStream stream = client.GetStream();
```

Объект `stream` типа `NetworkStream` инкапсулирует низкоуровневые сетевые функции и очень удобен для организации передачи/приема данных по сетевому каналу. Поток байтов данных, поступающих от клиента TCP, читается с помощью метода `Read` потока `stream` в буфер `buf`. Для отображения содержимого буфера данных на экране консоли нужно преобразовать его в формат строки `String`. Эту задачу выполняет оператор

```
s1 = System.Text.ASCIIEncoding.ASCII.GetString
     (buf, 0, bytesRead);
```

Данные, преобразованные в формат строки, отображаются на экране посредством оператора

```
Console.WriteLine("DATA RECEIVED: " + s1);
```

По окончании обработки данных объект `stream` и "рабочий" сокет `client` закрываются, и сетевой сеанс с клиентом заканчивается.

Исходный текст приложения-клиента TCP показан в листинге 4.24.

Листинг 4.24. Программный код приложения-клиента TCP

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO.Ports;

namespace TCPClient
{
  class Program
  {
   public static SerialPort serialPort1;
   public static Thread myThread;

   static void Main(string[] args)
    {
      serialPort1 = new SerialPort("COM4", 9600,
      Parity.None, 8, StopBits.One);
      serialPort1.Open();
      myThread = new Thread(ThreadProc);
      myThread.Start();
 }

  public static void ThreadProc()
   {
    serialPort1.DataReceived += new
     SerialDataReceivedEventHandler
    (serialPort1_DataReceived);
    while (true);
   }
```

```
   public static void serialPort1_DataReceived
   (object sender, SerialDataReceivedEventArgs e)
   {
    byte[] buf = new byte[1024];
    String s1 = serialPort1.ReadLine();

    IPAddress addr = IPAddress.Parse("127.0.0.1");
    IPEndPoint iep = new IPEndPoint(addr, 5151);
    TcpClient client = new TcpClient();
    client.Connect(iep);

    NetworkStream stream = client.GetStream();
    int bytesRead = System.Text.ASCIIEncoding.ASCII.GetBytes
                    (s1, 0, s1.Length, buf, 0);
    stream.Write(buf, 0, bytesRead);
    stream.Close();
    client.Close();
   }
 }
}
```

В клиентском приложении используется пространство имен `System.Net`, как и в серверном приложении, и, кроме того, `System.IO.Ports`. Прием данных с последовательного порта COM4 выполняется в отдельном потоке, функция `ThreadProc` которого запускает обработчик события `DataReceived` класса `SerialPort`. Функция-обработчик `serialPort1_DataReceived` события `DataReceived` собственно и выполняет основную работу.

При поступлении с последовательного порта данные записываются в строку `s1` функцией `ReadLine` класса `serialPort1`. Затем создается клиентский сокет `client`, через который и будет выполняться передача данных TCP-серверу:

```
TcpClient client = new TcpClient();
```

Как и в серверном приложении, сокет должен быть привязан к конкретному адресу и порту (объект `iep`).

В отличие от серверного приложения здесь не нужно прослушивать входящие соединения, поэтому клиентский сокет подключается к серверу методом `Connect`:

```
client.Connect(iep);
```

Для передачи данных по сети в программе создается объект потока `stream`, метод `Write` которого позволяет передать последовательность байтов TCP-серверу. Предварительно строка `s1`, принятая с COM-порта, преобразуется в поток байтов с помощью оператора

```
int bytesRead = System.Text.ASCIIEncoding.ASCII.GetBytes
                (s1, 0, s1.Length, buf, 0);
```

По окончании передачи данных объект потока `stream` и сокет `client` закрываются.

Как в клиентском, так и в серверном приложении (в случае их использования на разных хостах) следует указать конкретные IP-адреса и выбрать подходящие номера портов. Если в сети имеется сервер имен (DNS-сервер), то при небольшой модификации программного кода клиента и сервера можно использовать вместо IP-адресов URL.

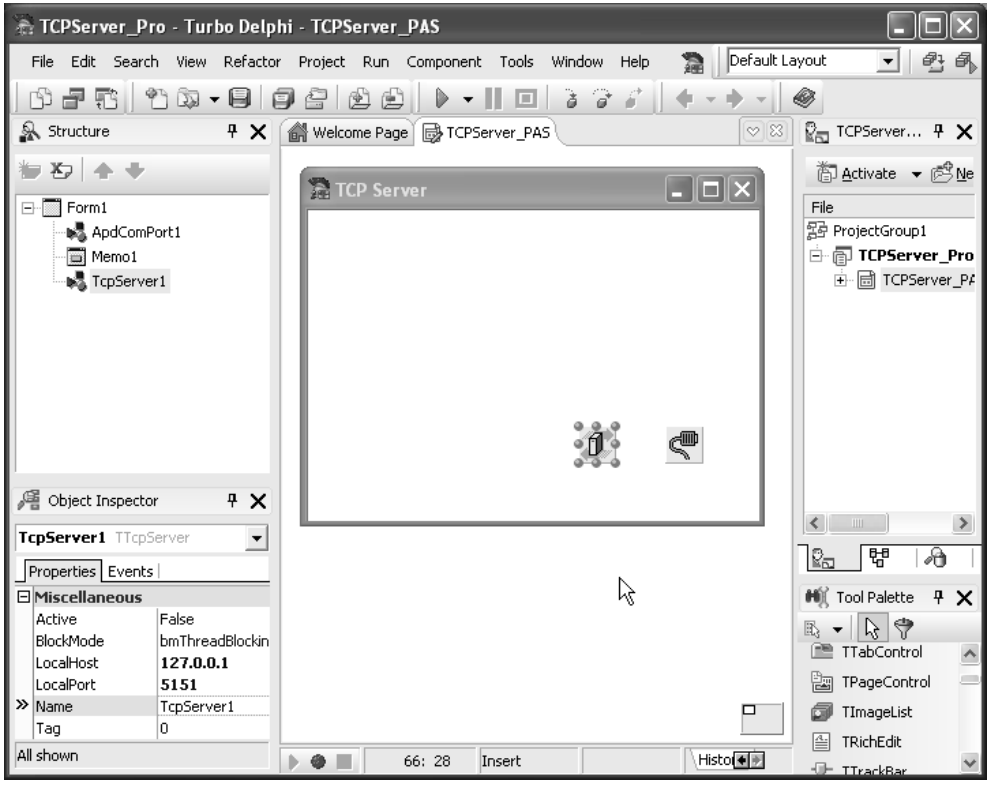


Рис. 4.35. Окно конструктора приложения сервера с настройками TCP-сервера

Решить задачу передачи данных с последовательного порта по сети в Delphi можно за счет применения компонента TApdComPort и компонетов палитры Internet. Как и в предыдущем примере, у нас будет два приложения: TCP-сервер и TCP-клиент. Для разнообразия разработаем сервер TCP, который будет отправлять строку данных, полученных с последовательного порта COM4, приложению-клиенту. Клиентское приложение будет принимать данные и отображать их в окне многострочного редактора.

На форме приложения сервера разместим экземпляры компонентов TTcpServer и TapdComPort (рис. 4.35).

Оба приложения (клиент и сервер) будут работать на одной машине, поэтому установим IP-адрес компонента TcpServer1 равным 127.0.0.1, а порту присвоим значение 5151. Кроме того, наш TCP-сервер будет работать в блокирующем режиме, поэтому свойство `BlockMode` оставим установленным по умолчанию. Сервер будет активизирован при запуске приложения, поэтому оставим свойство `Active` по умолчанию False. Свойствам компонента ApdComPort1 присвоим стандартные для порта COM4 значения (рис. 4.36).

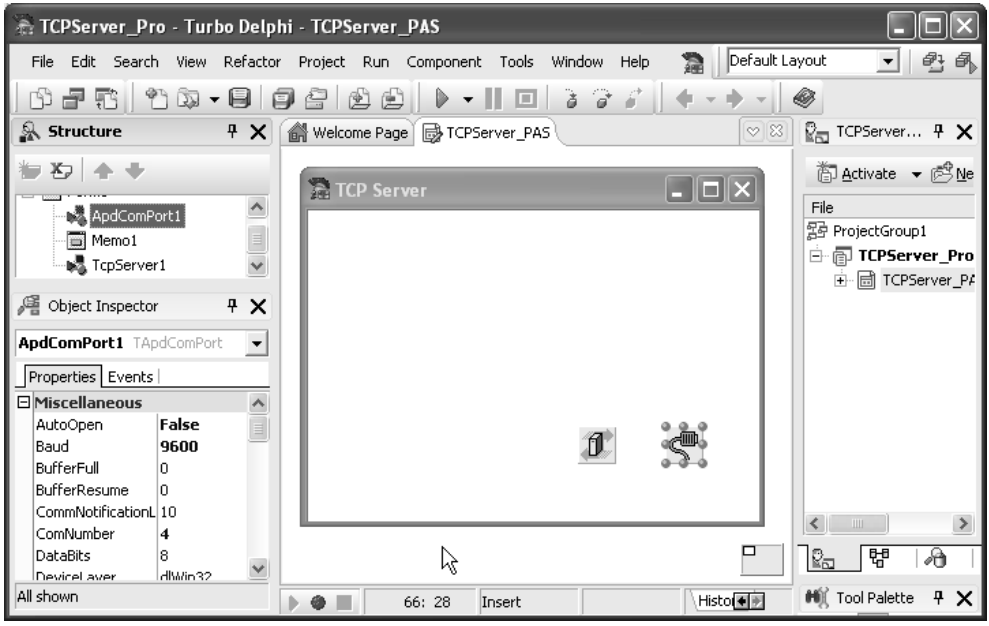


Рис. 4.36. Окно конструктора приложения с настройками компонента последовательного порта

Установим скорость обмена данными последовательного порта равной 9600 бод, количество битов в посылке данных равное 8, без контроля четности и с одним стоповым битом. Номер последовательного порта, с которым будет работать сервер, установим равным 4 (свойство `ComNumber`). Последовательный порт COM4 будет открываться при запуске приложения и закрываться по окончании его работы в соответствующих обработчиках событий формы приложения, поэтому свойство `AutoOpen` установим в значение False.

На форме приложения, кроме компонентов TTcpServer и TApdComPort, разместим экземпляр многострочного редактора TMemo, в окне которого будут отображаться полученные от порта COM4 данные.

Исходный текст приложения сервера TCP приведен в листинге 4.25.

Листинг 4.25. Исходный текст программы-сервера TCP

```
unit TCPServer_PAS;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
 Controls, Forms, Dialogs, StdCtrls, Sockets, OoMisc, AdPort;

type
  TForm1 = class(TForm)
    TcpServer1: TTcpServer;
    Memo1: TMemo;
    ApdComPort1: TApdComPort;
    procedure TcpServer1Accept(Sender: TObject; ClientSocket:
TCustomIpClient);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure ApdComPort1TriggerAvail(CP: TObject; Count: Word);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
  s1: String;
  sbuf: String;
implementation

{$R *.dfm}

procedure TForm1.ApdComPort1TriggerAvail(CP: TObject; Count: Word);
var
 I: Word;
 C: Char;
begin
  for I := 1 to Count do
   begin
    C:= ApdComPort1.GetChar();
    s1:= s1 + C;
    if C = #10 then
     begin
      Memo1.Lines.Append(s1);
      sbuf:= s1;
      s1:= '';
     end;
    end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 TcpServer1.Active:= False;
 ApdComPort1.Open:= False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 TcpServer1.Active:= True;
```

```
 ApdComPort1.Open:= True;
end;

procedure TForm1.TcpServer1Accept(Sender: TObject;
  ClientSocket: TCustomIpClient);
begin

  ClientSocket.Sendln (sbuf);
  ClientSocket.Close();
end;

end.
```

Основную работу в приложении выполняют две функции-обработчика. Первая, `TcpServer1Accept`, выбирается для события `OnAccept` серверного компонента (рис. 4.37).

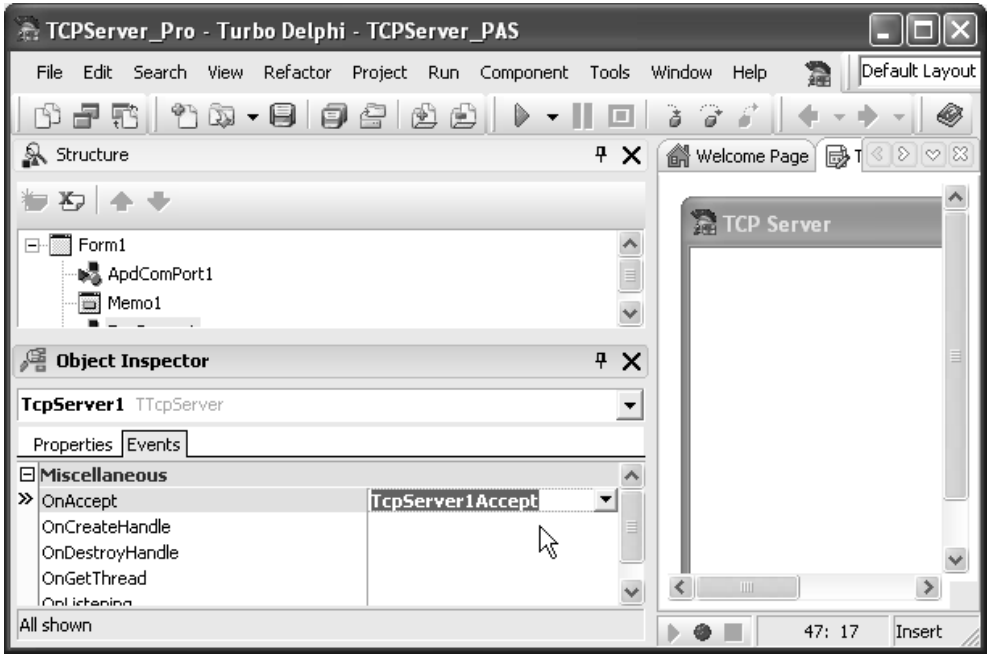


Рис. 4.37. Установка обработчика события `OnAccept` компонента TcpServer1

При вызове этой функции создается рабочий сокет `ClientSocket` для клиентского соединения, через которое с помощью метода `Sendln` содержимое буфера данных последовательного порта передается клиентскому приложению. По завершении передачи данных клиентский сокет закрывается с помощью оператора

```
ClientSocket.Close();
```

При этом прослушивающий сокет, созданный компонентом TcpServer1, будет ожидать последующих запросов на подключение.

Функция-обработчик `ApdComPort1TriggerAvail` компонента ApdComPort1 вызывается при возникновении события `OnTriggerAvail` (рис. 4.38).

Это событие генерируется в том случае, если в буфере приема последовательного порта имеются данные. В обработчике `ApdComPort1TriggerAvail` каждый байт данных помещается на следующую позицию в буфере `s1` с помощью операторов

```
C:= ApdComPort1.GetChar();
s1:= s1 + C;
```

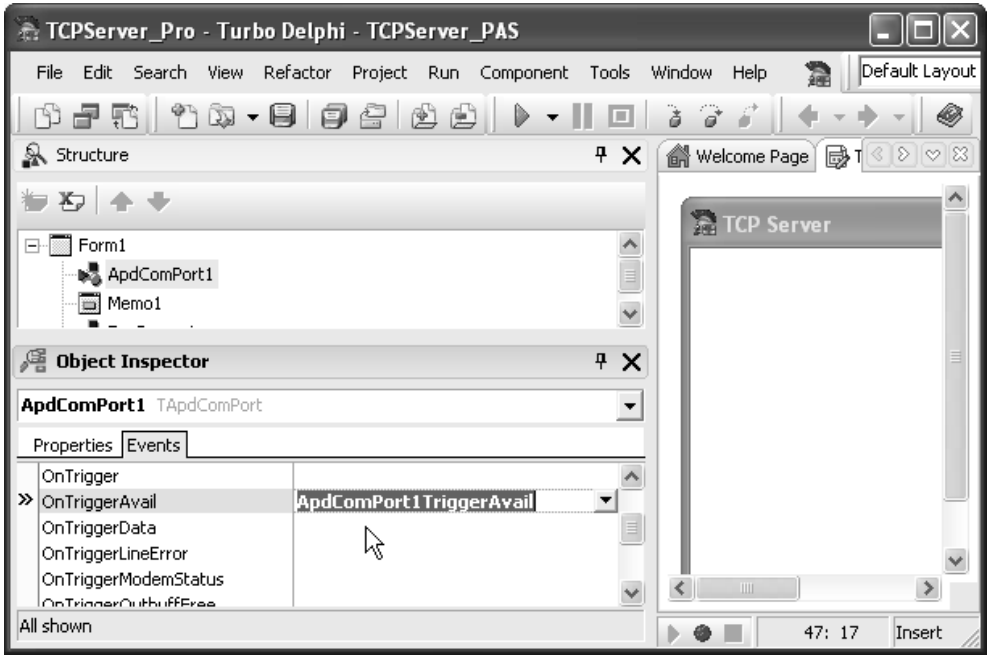


Рис. 4.38. Установка обработчика события `OnTriggerAvail` компонента ApdComPort1

При обнаружении во входном потоке символа возврата каретки содержимое буфера `s1` выводится в текстовое окно редактора Memo1, а данные копируются во вспомогательный буфер `sbuf`, откуда отправляются сетевому клиенту.

Приложение-клиент TCP содержит три компонента: клиент TCP (TTcpClient), многострочный редактор (TMemo) и кнопку (TButton) (рис. 4.39).

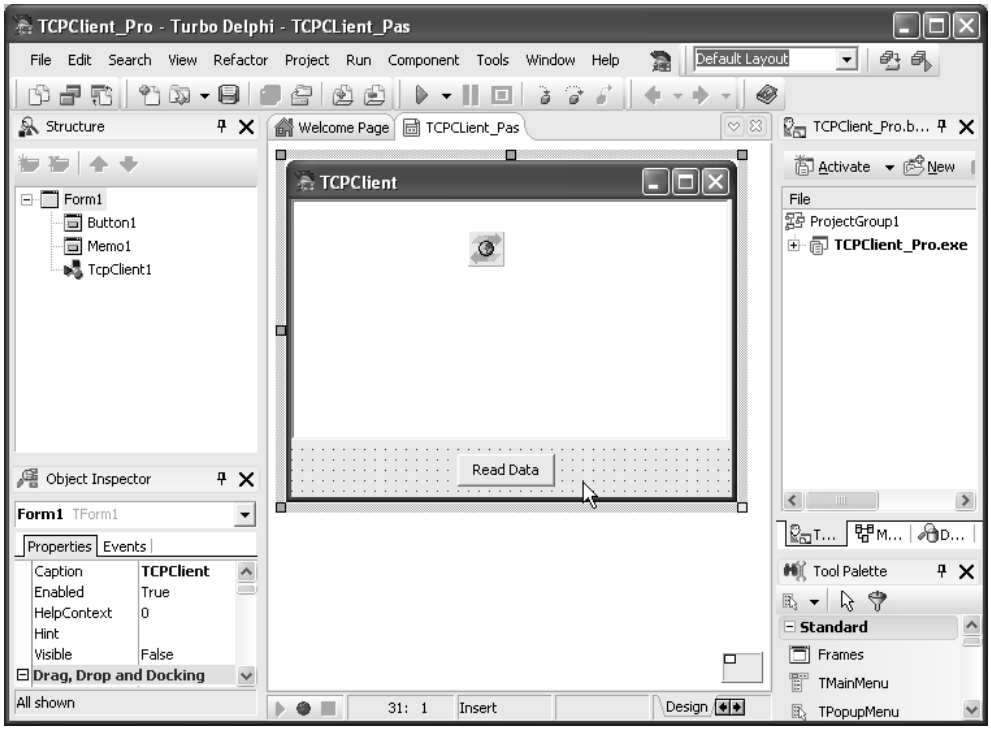


Рис. 4.39. Окно конструктора приложения с компонентами

Приложение-клиент TCP читает данные с сервера TCP и отображает их в окне многострочного редактора Memo1.

Исходный текст клиентского приложения TCP приведен в листинге 4.26.

Листинг 4.26. Исходный текст приложения-клиента TCP

```
unit TCPClient_Pas;

interface
```

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
 Controls, Forms, Dialogs, StdCtrls, Sockets;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    TcpClient1: TTcpClient;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  s1: String;
implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
   if TcpClient1.Connected = False  then
     TcpClient1.Connect();
   s1:= TcpClient1.Receiveln();
   TcpClient1.Disconnect();
   Memo1.Lines.Add(s1);
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
```

```
 TcpClient1.Active:= False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  TcpClient1.Active:= True;
end;

end.
```

Компонент TTcpClient активизируется при запуске приложения (событие `FormCreate`) и останавливается по завершении работы приложения (событие `FormClose`). Для экземпляра компонента `TCPClient1` нужно установить IP-адрес удаленного хоста (свойство `RemoteHost`) в 127.0.0.1 и значение порта (свойство `RemotePort`), равное 5151.

Чтение данных с TCP-сервера осуществляется при нажатии кнопки **Read Data** (обработчик `Button1Click`). В отличие от сервера, клиенту TCP не нужно прослушивать соединения, поэтому он выполняет запрос на соединение (если отсутствует) с сервером с помощью функции `Connect`:

```
TcpClient1.Connect();
```

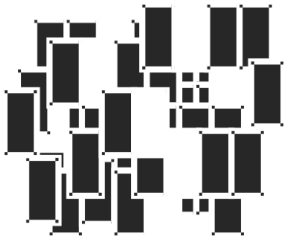
После установления соединения можно принимать данные от сервера, что выполняет оператор

```
s1:= TcpClient1.Receiveln();
```

Строка данных помещается в переменную `s1` типа `String` и выводится в окно редактора Memo1.

Следует обратить ваше внимание на то, что при разработке демонстрационных приложений мы не учитывали возможных сбойных ситуаций, которые могут возникать в процессе обмена данными, например, отсутствие несущей при работе с RS-232, или отсутствие сетевого соединения между клиентом и сервером. В реальных разработках нужно обязательно обрабатывать подобные ситуации, используя соответствующие программные механизмы.

Глава 5



Программирование последовательного интерфейса в Linux

Для анализа функционирования последовательного интерфейса в операционной системе Linux вкратце ознакомимся с базовыми принципами, по которым осуществляется взаимодействие системы и внешних устройств, к которым относятся и коммуникационные каналы. В операционной системе UNIX повсеместно используется понятие "файл" или "объект файловой системы", которое здесь трактуется в более широком смысле, чем в других операционных системах. Начнем с того, что в операционных системах UNIX все объекты, включая текстовые и двоичные дисковые файлы, именованные и неименованные каналы, сетевые сокеты и устройства, обрабатываются как объекты файловой системы.

Файловая система UNIX представляет собой иерархически организованную структуру и обеспечивает унифицированный интерфейс доступа к данным на различных носителях, в том числе и на периферийных устройствах. Для всех объектов файловой системы устанавливаются и контролируются права доступа к файлам, выполняются операции создания, удаления, записи и чтения данных.

Система управляет объектами файловой системы при помощи унифицированного набора системных функций ядра, к которым могут обращаться пользовательские процессы посредством системных вызовов, таких, например, как `open()`, `read()`, `write()` и `close()`. Все файловые операции программы пользователя выполняются одинаково как для дисковых файлов, так и для специальных файлов устройств.

Файлы устройств позволяют операционной системе UNIX и другим программам взаимодействовать с аппаратными средствами и периферийными устройствами системы. Здесь я хочу сделать важное замечание: *нужно отличать файлы устройств от драйверов устройств*. Управление конкретным физическим устройством осуществляется посредством специальной программы, которая называется *драйвером устройства*. *Файлы устройств* сами по себе не являются драйверами, их можно представить как шлюзы, через которые драйвер получает запросы.

Когда ядро получает запрос к файлу устройства, оно просто передает этот запрос соответствующему драйверу. Таким образом, файлы устройств позволяют программам взаимодействовать с драйверами ядра. По своей структуре они не являются файлами данных, но обрабатываются базовыми средствами файловой системы, а их характеристики записываются на диск. Взаимодействие пользовательской программы, файла устройства и драйвера показано на рис. 5.1. Здесь пользовательская программа обращается к последовательному порту, подсоединенному к параллельному порту (ему может соответствовать файл устройства /dev/ttyS0).

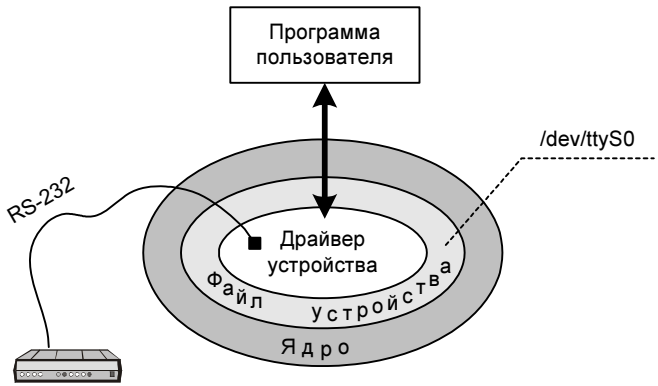


Рис. 5.1. Взаимодействие программы пользователя с устройством в UNIX

Файлы устройств, в свою очередь, можно разделить на два типа:

- файлы байт-ориентированных устройств;
- файлы блок-ориентированных устройств.

Файлы байт-ориентированных устройств позволяют связанным с ними драйверам выполнять свою собственную буферизацию ввода/вывода. *Файлы*

блок-ориентированных устройств обрабатываются драйверами, которые осуществляют ввод-вывод большими блоками и возлагают обязанности по выполнению задач буферизации на ядро. Некоторые типы аппаратных средств, такие как накопители на жестких дисках (винчестеры) и магнитных лентах, могут быть представлены файлами любого типа.

В системе может присутствовать несколько однотипных устройств, поэтому файлы устройств характеризуются двумя номерами: старшим и младшим. Старший номер устройства информирует ядро, к какому драйверу относится данный файл, а младший номер сообщает драйверу, к какому физическому устройству следует обращаться. Например, старший номер устройства 4 в Linux обозначает драйвер последовательного порта. Первый последовательный порт (/dev/ttyS0) будет иметь старший номер 4 и младший номер 0.

Некоторые драйверы используют младший номер устройства нестандартным способом. Например, драйверы накопителей на магнитных лентах часто руководствуются им при выборе плотности записи, а также определяют, нужно ли перемотать ленту после закрытия файла устройства. В некоторых системах драйвер терминала, управляющий последовательными устройствами, использует младшие номера устройств для идентификации модемов.

В современных файловых системах UNIX определен интерфейс уровня ядра, позволяющий работать с различными аппаратными интерфейсами. При этом часть файлов обрабатывается традиционной дисковой подсистемой, другие управляются отдельными драйверами ядра, как в случае сетевых файловых систем, где используется драйвер, перенаправляющий запросы серверу на другой компьютер.

Поскольку все устройства UNIX доступны посредством файлов устройств, то для них действительны все операции, выполняемые над файлами. Тем не менее, существует и определенная специфика работы с устройствами, которую необходимо учитывать при различных манипуляциях с этими объектами.

Все устройства операционной системы образуют так называемую *подсистему ввода-вывода*, связывающую работающие процессы с периферийными устройствами, такими как накопители на магнитных дисках и лентах, терминалы, принтеры и сети, с одной стороны, и с модулями ядра, которые управляют устройствами и именуются драйверами устройств, с другой. Драйверы устройств управляют каким-либо одним типом устройств — в системе может быть один дисковый драйвер для управления всеми дисководами, один терминальный драйвер для управления всеми терминалами и один ленточный драйвер для управления всеми ленточными накопителями.

Если в системе имеются однотипные устройства от разных изготовителей, например, две марки ленточных накопителей, то они воспринимаются системой как устройства двух различных типов, что требует наличия двух отдельных драйверов из-за различных управляющих команд для этих устройств.

Кроме того, система поддерживает так называемые *виртуальные устройства*, с которыми не связано ни одно конкретное физическое устройство. Примером такого устройства может выступать физическая память. В этом есть определенный смысл, поскольку позволяет процессу обращаться к памяти как к устройству. Команда `ps`, например, обращается к информационным структурам ядра в физической памяти, чтобы сообщить статистику процессов.

Операционная система UNIX поддерживает два типа устройств — устройства ввода-вывода блоками и устройства неструктурированного или посимвольного ввода-вывода. *Устройства ввода-вывода блоками* или *блочные устройства ввода-вывода*, такие как диски и ленты, для остальной части системы выглядят как запоминающие устройства с произвольной выборкой. *Устройства посимвольного ввода-вывода* манипулируют с произвольным потоком данных в виде последовательности байтов. К ним относятся все остальные устройства, в том числе терминалы и сетевое оборудование. Последовательные порты также относятся к устройствам посимвольного ввода-вывода.

Пользовательские программы взаимодействуют с устройствами посредством специальных файлов устройств, которые занимают определенное место в иерархии каталогов файловой системы. Файл устройства отличается от других файлов типом, хранящимся в его индексе, в зависимости от устройства (блочное или символьное), которое он представляет. В тех случаях, когда устройство имеет как блочный, так и символьный интерфейс, для него определены два файла: специальный файл блочного устройства ввода-вывода и специальный файл устройства посимвольного ввода-вывода.

Файлы устройств, так же как и обычные файлы, управляются системными вызовами `open()`, `close()`, `read()` и `write()`.

Файлы устройств в UNIX обозначаются специальным образом. Например, файл /dev/console соответствует терминалу консоли. Вывод, направленный в специальный файл /dev/console, будет появляться на экране терминала, а при попытке прочитать данные из файла /dev/console фактически будет прочитан символ с клавиатуры. Файл первого последовательного порта обозначается как /dev/ttyS0. Некоторые файлы устройств имеют особое назначение и ссылаются на сервисы операционной системы, например, /dev/null, /dev/random.

Для создания файлов устройств существует команда `mknod`, в которой указывается тип файла (блочный или символьный), а также старший и младший номера устройства. Создавать файл устройства может только суперпользователь root. Команда `mknod` имеет такой синтаксис:

```
mknod [опции] имя {bc} старший_номер младший_номер
mknod [опции] имя p
```

Имя устройства должно быть задано в форме /dev/dev_name. Тип устройства указывается символом `c` (для символьных устройств с побайтовым доступом, как, например, последовательные порты) или символом `b` (для блочных устройств, с которыми возможен обмен данными блоками, например, для дисковых накопителей).

Следующий параметр представляет собой старший номер уникального идентификатора, характеризующего группу устройств (например, 4 — идентификатор виртуальных терминалов).

После старшего номера указывают младший номер устройства, представляющий собой идентификатор конкретного устройства в данной группе (например, младший номер 0 в группе старшего номера 4 — идентификатор первой системной виртуальной консоли, а 63 — идентификатор последней теоретически возможной из них).

Старший номер устройства показывает его тип, которому соответствует точка входа в таблице ключей устройств, *младший номер устройства* — это порядковый номер единицы устройства данного типа. С одним старшим номером устройства может быть связано множество периферийных устройств, а младший номер устройства позволяет отличить их одно от другого. Не нужно создавать специальные файлы устройств при каждой загрузке системы; их достаточно корректировать, если изменилась конфигурация системы, например, если к установленной конфигурации были добавлены устройства.

Кроме указанных, команда `mknod` может принимать и дополнительные опции GNU:

```
[-m права] [--help] [--version] [--]
```

В следующем примере создается устройство /dev/ttyS5:

```
# mknod /dev/ttyS c 4 5
```

Здесь параметр `c` указывает на тип устройства (в данном случае, символьное), 4 — старший номер устройства, а 5 — младший номер устройства.

Фактически команда mknod создает *именованный канал* — специальный файл символьного или блочного устройства с указанным именем. Созданный специальный файл не занимает места на диске и используется только для взаимодействия с операционной системой, но не для хранения данных.

GNU-версия mknod позволяет считать символ u синонимом типа c. Рассмотрим дополнительную опцию GNU, обозначаемую как m или --mode. Эта опция определяет права доступа к создаваемым файлам. Опция может быть указана в одной из двух форм:

```
-m права
--mode=права
```

Значение прав доступа к создаваемым файлам становится равным по величине значению аргумента права; оно может иметь как символьную форму, описанную в man-странице команды chmod, так и восьмеричную.

Пример использования команды mknod с указанием атрибутов доступа:

```
$ mknod --mode=644 /dev/ttyS6 c 4 6
```

Эта команда создаст файл устройства для последовательного порта с младшим номером 6 и атрибутами доступа 644.

Операционная система UNIX, помимо стандартных средств командной оболочки для работы с объектами файловой системы, включает целый ряд функций ядра, доступных через системные вызовы, для работы с файлами.

Функции прикладного интерфейса программирования (API), представляющие собой системные вызовы операционной системы UNIX, дают возможность пользователям, знакомым с языком C или C++, манипулировать с файлами в своих программах. Кроме того, что очень важно, знание механизмов работы таких функций способствует более глубокому пониманию работы операционной системы в целом и правильному использованию ее возможностей. Необходимо учитывать и то, что все команды операционной системы (chmod, chown, cp, mv и т. д.), выполняющие манипуляции с файлами и доступные пользователю, созданы на основе системных вызовов, выполняющих обработку файлов на низком уровне.

Использование функций прикладного интерфейса программирования рассмотрим на примерах программирования обмена данными через последовательный порт операционной системы Linux, но вначале познакомимся с методами базовых настроек этого коммуникационного ресурса.

5.1. Настройка последовательного интерфейса

В общем случае настройку последовательного интерфейса в операционной системе Linux выполняют в несколько этапов. На первом этапе нужно определить, какие коммуникационные ресурсы доступны. Это можно выполнить с помощью команды `dmesg`, например:

```
# dmesg | grep ttyS*
ttyS0 at 0x03f8 (irq = 4) is a 16550A
```

Большинство операционных систем Linux включает один из популярных графических интерфейсов, таких как *Gnome* или *KDE*, для которых разработаны удобные графические приложения для настройки, диагностики и мониторинга ресурсов. Например, конфигурацию устройств в Linux с загруженной оболочкой KDE можно просмотреть, используя программу *KInfo* (рис. 5.2).

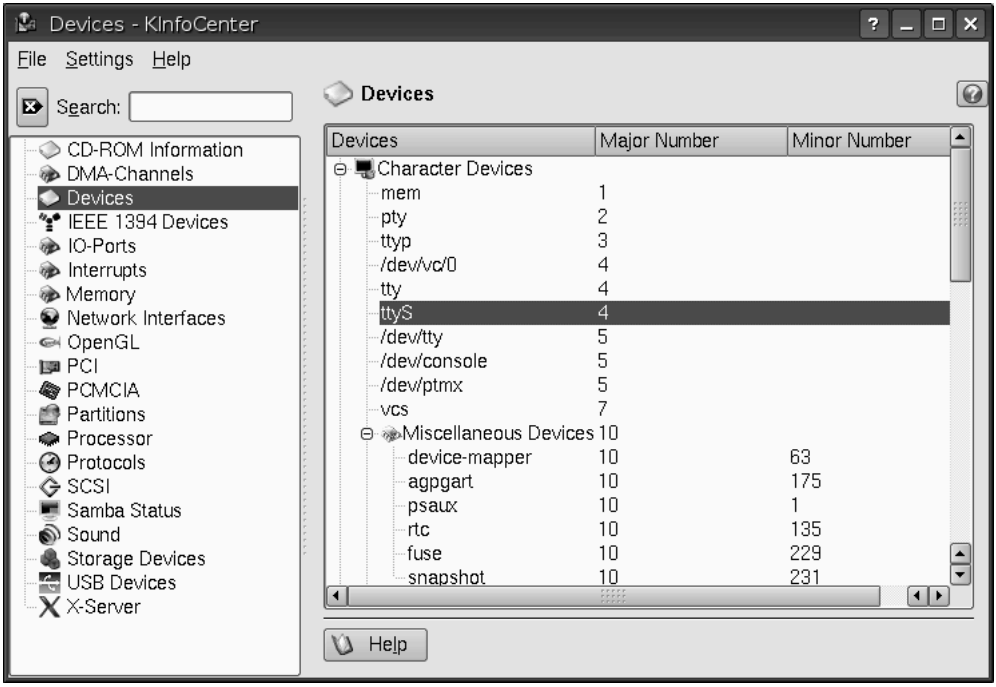


Рис. 5.2. Вывод конфигурации устройств в графической оболочке KDE

Для настройки конкретного выбранного последовательного интерфейса можно воспользоваться одной из стандартных утилит setserial или minicom.

5.1.1. Программа setserial

Программа setserial включается во все версии операционных систем Linux и используется для получения информации о конфигурации последовательных интерфейсов операционной системы, а также для установки параметров последовательных портов. При начальной загрузке операционной системы инициализируются порты COM1—COM4, при этом им присваиваются аппаратные ресурсы (адреса портов ввода-вывода и номера прерываний) по умолчанию. С помощью утилиты setserial можно изменить параметры конфигурации. Программа имеет следующий синтаксис:

```
setserial [ -abqvVWz ] device [ parameter1 [ arg ] ] ...
setserial -g [ -abGv ] device1 ...
```

Настройка параметров последовательного порта происходит при загрузке операционной системы. Утилита setserial вызывается, как правило, из командного скрипта rc.local, который размещен в каталоге /etc/rc.local. Параметр `device` в списке параметров команды setserial имеет вид /dev/ttySx, где `x` находится в диапазоне 0—3. Для конфигурирования последовательного порта в Linux требуются, как правило, права суперпользователя root, хотя некоторые параметры могут быть настроены и обычным пользователем. В любом случае нужно просмотреть man-страницу данной команды.

Если команда вводится без параметров, то на экран консоли выводятся базовые характеристики последовательного порта (тип микросхемы UART (8250A, 16450, 16550, 16550 и т. д.)), адрес базового порта последовательного интерфейса, номер присвоенной линии прерывания и состояние некоторых флагов управления и состояния.

Вот смысл некоторых опций команды setserial:

- `-g` — выводятся базовые характеристики указанного последовательного интерфейса;
- `-a` — выводится подробная информация о конфигурации последовательного порта;
- `-b` — выводится суммарная информация об устройстве, обычно используется в процессе начальной загрузки системы при выполнении скрипта rc.serial;

- `-G` — выводит информацию о конфигурации в формате, поля которого можно использовать в качестве параметров командной строки при настройке последовательного порта;
- `-v` — выводит дополнительную информацию о состоянии последовательного порта;
- `-V` — выводит информацию о версии программы;
- `-z` — предварительно очищает все флаги состояния перед их повторной установкой.

Утилита setserial позволяет настроить последовательный порт с помощью установки соответствующих параметров. Все значения параметров предполагаются в десятичном формате, а для установки шестнадцатеричного значения нужно использовать префикс "0x". Вот смысл некоторых параметров программы setserial:

- `port` *`номер_порта`* — позволяет установить номер порта;
- `irq` *`номер_прерывания`* — позволяет назначить линию прерывания IRQ;
- `uart` *`тип_устройства`* — позволяет указать тип устройства UART, используемого в последовательном интерфейсе. Допустимо задавать типы: none, 8250, 16450, 16550, 16550A, 16650, 16650V2, 16654, 16750, 16850, 16950, 16954. Если указано значение "none", то порт отключается. Этот параметр часто требуется при задании типа асинхронного приемопередатчика (UART) для PCI-плат расширения, которые могут не идентифицироваться утилитой setserial. Обычно тип UART таких устройств устанавливается как "unknown". Перед установкой типа микросхемы асинхронного приемопередатчика платы расширения интерфейса следует детально изучить техническую документацию на такие устройства;
- `autoconfig` — при указании этого параметра программа setserial запрашивает ядро для выполнения автоматического конфигурирования последовательного порта. Для корректной настройки необходимо правильно указать адрес базового порта ввода-вывода. В этом случае ядро попытается определить тип UART, а при указании параметра `auto_irq` попытается присвоить устройству номер линии прерывания IRQ. Параметр `autoconfig` можно задавать только после того, как указан номер порта (`port` *`номер_порта`*), автоматическое присвоение линии прерывания (`auto_irq`) и пропуск тестирования асинхронного приемопередатчика (`skip_test`);

- `auto_irq` — при задании параметра автоконфигурирования (`autoconfig`) выполняется попытка назначения линии прерывания устройству. В целом ряде случаев правильный результат такой настройки не гарантируется. Лучше всего избегать использования этого параметра, а вместо него настроить прерывание, задав явным образом номер IRQ в параметре `irq`;
- `^auto_irq` — запрещает определять линию прерывания IRQ для устройства при автоконфигурировании;
- `skip_test` — указывает на необходимость пропуска тестирования UART. В некоторых случаях пропуск тестирования типа асинхронного приемопередатчика оказывается полезным. Дело в том, что в целом ряде устройств используются микросхемы UART, которые не полностью совместимы с классическими типами таких устройств, например, в них может отсутствовать режим внутренней диагностики при замыкании выхода на вход (loopback diagnostic). Пропуск тестирования в таких случаях позволит ядру Linux правильно сконфигурировать устройство;
- `^skip_test` — указывет на обязательное выполнение тестирования UART при автоконфигурировании;
- `baud_base` *`скорость_обмена`* — позволяет установить скорость обмена, которая вычисляется делением тактовой частоты кристалла микросхемы UART на 16. Обычно это значение равно максимальной для UART скорости 115200 бод;
- `spd_hi` — позволяет работать на скорости 57,6 Кбод для приложений, которые настроены на скорость 38,4 Кбод. Этот параметр может установить непривилегированный пользователь;
- `spd_vhi` — позволяет работать на скорости 115 Кбод для приложений, которые настроены на скорость 38,4 Кбод. Этот параметр может установить непривилегированный пользователь;
- `spd_shi` — позволяет работать на скорости 230 Кбод для приложений, которые настроены на скорость 38,4 Кбод. Этот параметр может установить непривилегированный пользователь;
- `spd_warp` — позволяет работать на скорости 460 Кбод для приложений, которые настроены на скорость 38,4 Кбод. Этот параметр может установить непривилегированный пользователь;
- `spd_cust` — позволяет установить пользовательское значение коэффициента деления тактовой частоты для приложений, которые настроены на скорость 38,4 Кбод. В этом случае скорость обмена будет определяться

как результат деления параметра `baud_base` на параметр `spd_cust`. Этот параметр может установить непривилегированный пользователь;

- `spd_normal` — позволяет работать на скорости 38,4 Кбод для приложений, которые настроены на скорость 38,4 Кбод. Этот параметр может установить непривилегированный пользователь;
- `divisor` *`коэффициент_деления`* — позволяет установить пользовательское значение коэффициента деления. Параметр должен применяться, если выбран режим `spd_cust` и последовательный порт настроен на скорость обмена 38,4 Кбод. Этот параметр может установить непривилегированный пользователь;
- `close_delay` *`задержка`* — устанавливает интервал времени (в сотых долях секунды), в течение которого сигнал на линии DTR должен оставаться на низком уровне после отключения вызывающего устройства перед повторной установкой этого сигнала в высокий уровень. Значение по умолчанию для этого параметра принимается равным 50 (0,5 сек);
- `closing_wait` *`задержка`* — устанавливает интервал времени (в сотых долях секунды), в течение которого ядро должно ожидать передачи данных через последовательный порт перед закрытием порта. Если в качестве параметра указано "none", задержка не происходит. Если указано "infinite", ядро будет "бесконечно" ожидать завершения обмена данными. Значение по умолчанию этого параметра равно 3000 (30 сек). Если указано слишком большое значение, то последовательный порт может "зависнуть" на длительное время в случае, когда порт закрывается при обрыве соединения, а в буфере приемопередатчика имеются данные. При выборе небольшого интервала ожидания возможна потеря данных, особенно при передаче данных от сравнительно медленного устройства, например, плоттера. Этот параметр следует выбирать с учетом технических характеристик конкретного оборудования, участвующего в обмене данными.

В этом списке перечислены важнейшие параметры установки последовательного порта, используемые утилитой setserial. Более подробную информацию обо всех параметрах команды можно почерпнуть из man-страниц.

Утилита setserial не выполняет конфигурирование аппаратных настроек последовательного порта, она сообщает ядру Linux только возможные значения этих параметров. В Linux последовательным портам соответствуют устройства /dev/ttyS*, которые, в свою очередь, по аналогии с MS-DOS, соответствуют COM-портам (табл. 5.1).

Таблица 5.1. *Соответствие последовательных портов MS-DOS и Linux*

Устройство в Linux	Устройство MS-DOS	Аппаратные ресурсы
/dev/ttys0	COM1	Порт 0x3F8, IRQ = 4
/dev/ttys1	COM2	Порт 0x2F8, IRQ = 3
/dev/ttys2	COM3	Порт 0x3E8, IRQ = 4
/dev/ttys3	COM4	Порт 0x2E8, IRQ = 3

Как правило, последовательные порты используют линии 3 и 4 прерывания. Выделение дополнительных линий прерывания может оказаться затруднительным, хотя можно применить и уже задействованные системой линии прерывания. Например, в операционной системе прерывания с номерами 5 и 7 обычно выделяются параллельным портам LPT2 и LPT1 соответственно. Если один или оба принтера не применяются, то можно задействать эти линии прерывания в последовательных портах. Такой вариант может пригодиться при использовании модулей расширения последовательного порта, для которых может не оказаться свободных линий прерывания.

Примеры использования утилиты setserial приведены далее.

```
# setserial -a /dev/ttyS0
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
        Baud_base: 115200, close_delay: 50, divisor: 0
        closing_wait: 3000
        Flags: spd_normal skip_test
# setserial -a /dev/ttyS1
/dev/ttyS1, Line 1, UART: unknown, Port: 0x02f8, IRQ: 3
        Baud_base: 115200, close_delay: 50, divisor: 0
        closing_wait: 3000
        Flags: spd_normal skip_test
```

Здесь выполняется вывод информации о параметрах последовательных портов /dev/ttyS0 и /dev/ttyS1 (COM1 и COM2 в аббревиатуре MS-DOS) с помощью параметра `-a`.

Первые два последовательных порта /dev/ttyS0 и /dev/ttyS1 присутствуют в аппаратной части по умолчанию и конфигурируются системой автоматически. Отдельные настройки этих портов можно изменить, указав определенные значения рассмотренных ранее параметров, но следует учитывать, что

при некорректном конфигурировании доступ к портам может оказаться заблокированным.

Немного сложнее выглядит картина с настройкой PCI-модулей расширения ввода-вывода. Для них выделяются, как правило, файлы устройств /dev/ttyS2 и /dev/ttyS3 (COM3 и COM4 в аббревиатуре MS_DOS). При этом система может не сконфигурировать надлежащим образом все параметры устройства. Например, для модуля расширения NetMos с двумя дополнительными последовательными портами команда setserial выведет следующую информацию:

```
# setserial -a /dev/ttyS2
/dev/ttyS2, Line 2, UART: unknown, Port: 0x03e8, IRQ: 4
        Baud_base: 115200, close_delay: 50, divisor: 0
        closing_wait: 3000
        Flags: spd_normal skip_test

# setserial -a /dev/ttyS3
/dev/ttyS3, Line 3, UART: unknown, Port: 0x02e8, IRQ: 3
        Baud_base: 115200, close_delay: 50, divisor: 0
        closing_wait: 3000
        Flags: spd_normal
```

Здесь не был определен тип устройства UART (unknown), поэтому необходимо либо задать режим автоконфигурирования, либо установить тип устройства вручную с помощью параметра `uart`.

Вывод данных при задании опции `-G` для порта /dev/ttyS0 имеет вид:

```
# setserial -G /dev/ttyS0
/dev/ttyS0 uart 16550A port 0x03f8 irq 4 baud_base 115200 spd_normal
skip_test
```

Далее рассмотрим несколько примеров установки параметров обмена для порта /dev/ttyS0. Для установки параметров в командной строке после имени утилиты следует указать имя устройства, а далее тип и значение параметра. В следующих командных строках выполняется установка скорости обмена (`baud_base`), равной 9600 бод, и проверка установленного значения:

```
# setserial /dev/ttyS0 Baud_base 9600
# setserial -a /dev/ttyS0
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
        Baud_base: 9600, close_delay: 50, divisor: 0
        closing_wait: 3000
        Flags: spd_normal skip_test
```

С помощью следующей командной строки устанавливается тип устройства UART для порта /dev/ttyS0, совместимый с 8250:

```
# setserial /dev/ttyS0 uart 8250
```

Подобную команду можно использовать для настройки PCI-плат расширения, для которых при конфигурировании не определен тип асинхронного приемопередатчика.

5.1.2. Программа minicom

Другим популярным приложением, используемым для конфигурирования последовательного порта в операционных системах Linux, является утилита minicom, позволяющая выбрать и установить параметры последовательного обмена данными с помощью текстового окна. Это приложение является многофункциональным и позволяет настроить параметры последовательных интерфейсов обмена данными, режим dial-up, параметры автодозвона, реализовать сценарии доступа нескольких пользователей к коммуникационным ресурсам и обмен данными с помощью скриптов, запускаемых с помощью специального встроенного интерпретатора, и т. д.

Программа minicom имеет следующий синтаксис:

```
minicom [-somMlwz8] [-c on|off]
        [-S script] [-d entry]
        [-a on|off] [-t term]
        [-p pty] [-C capturefile] [configuration]
```

В простейшем варианте утилита minicom запускается с командной строки без параметров, при этом используются настройки по умолчанию. При запуске программа ищет переменную окружения `MINICOM` с установленными параметрами.

Вот смысл наиболее часто используемых опций команды minicom:

- `-s` — установка программы. При указании этой опции редактируются настройки по умолчанию, сохраненные в файле конфигурации /etc/minirc.dfl. Эта опция полезна при первом запуске утилиты, а также в тех случаях, когда по каким-либо причинам minicom не запускается в системе. По этим причинам многие Linux-системы уже содержат предварительно скомпилированный файл конфигурации;
- `-o` — при запуске программы инициализация не выполняется, а код инициализации пропускается. Опция полезна при возобновлении работы в предыдущей сессии без сброса установок;

- `-t` *`тип_терминала`* — при указании этого флага настройки, хранящиеся в переменной `TERM`, будут перезаписаны в соответствии со значениями в переменной среды `MINICOM`;
- `-S` *`имя_скрипт`* — при запуске выполняет скрипт с указанным именем;
- `configuration` — имя файла конфигурации, который будет использоваться вместо заданного по умолчанию minirc.dfl. Предполагается, что задаваемый таким параметром файл имеет имя minirc.configuration. Применение этой опции позволяет запускать программу с различными файлами конфигурации, настроенными для разных коммуникационных устройств и разных пользователей.

Файлы настройки программы minicom хранятся в одном из каталогов /var/lib/minicom, /usr/local/etc или /etc. Для определения каталога в командной строке вводится команда

```
# minicom -h
```

В этих же каталогах хранятся примеры командных скриптов и таблицы преобразования.

Рассмотрим пример использования программы minicom для настройки скорости обмена данными. Запустим утилиту minicom без параметров (используется файл конфигурации minirc.dfl):

```
# minicom
```

При запуске утилиты появляется окно, показанное на рис. 5.3.

Для перехода в основное меню программы следует нажать комбинацию клавиш <Ctrl>+<A>, а затем <Z> (рис. 5.4).

Для навигации по меню нужно использовать клавиши <↑> (вверх), <↓> (вниз), <←> (влево), <→> (вправо). Для подтверждения выбора (Choose) нужно использовать клавишу<Enter>, а для отмены действия (Cancel) — клавишу <Esc>.

Программа minicom имеет много опций, позволяющих устанавливать различные параметры последовательного порта, модемного соединения и протокола обмена данными. Кроме того, с ее помощью можно выполнять передачу и прием файлов, а также некоторые диагностические процедуры. В данном случае ограничимся настройкой скорости обмена через последовательный порт, выбрав ее равной 9600 бод. Для этого из списка опций меню выберем **cOnfigure Minicom**, нажав клавишу <O>. В раскрывшемся окне (рис. 5.5) выберем опцию **Serial port setup**.

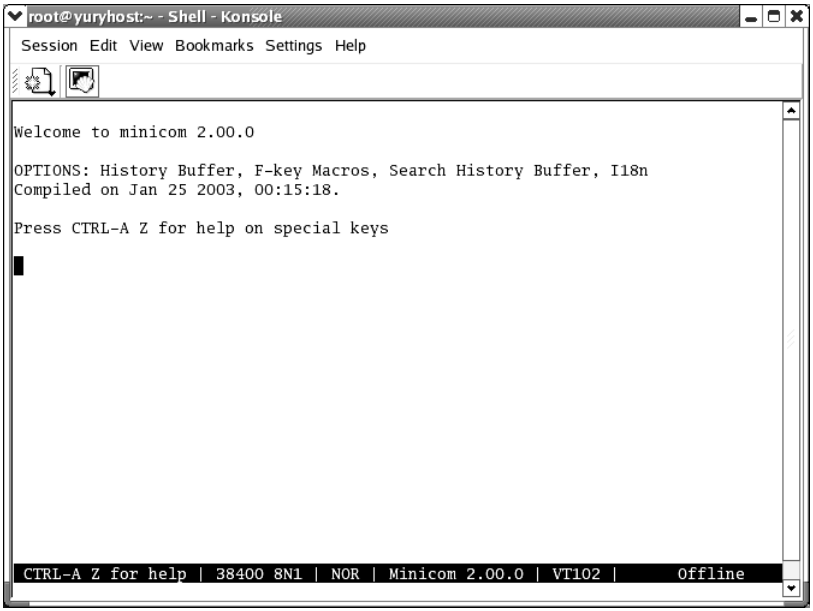


Рис. 5.3. Окно запуска программы minicom

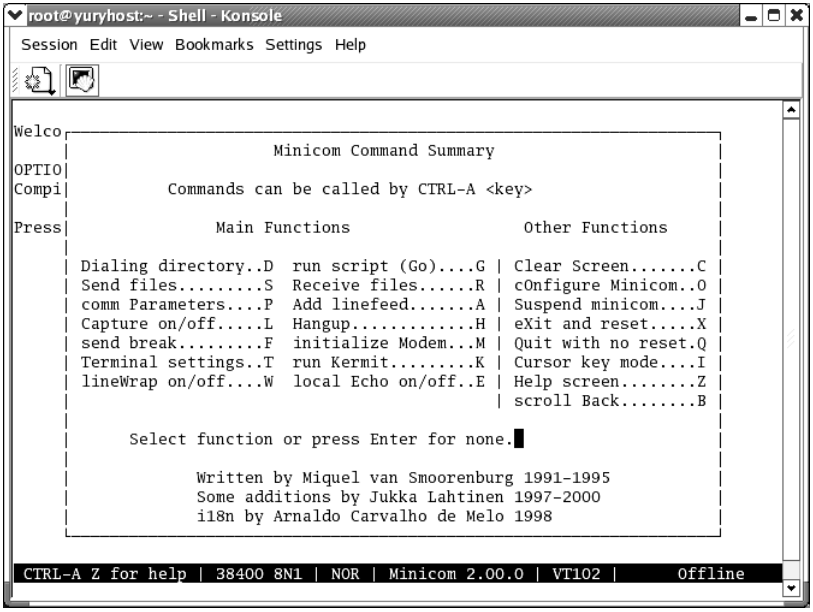


Рис. 5.4. Меню выбора команд утилиты minicom

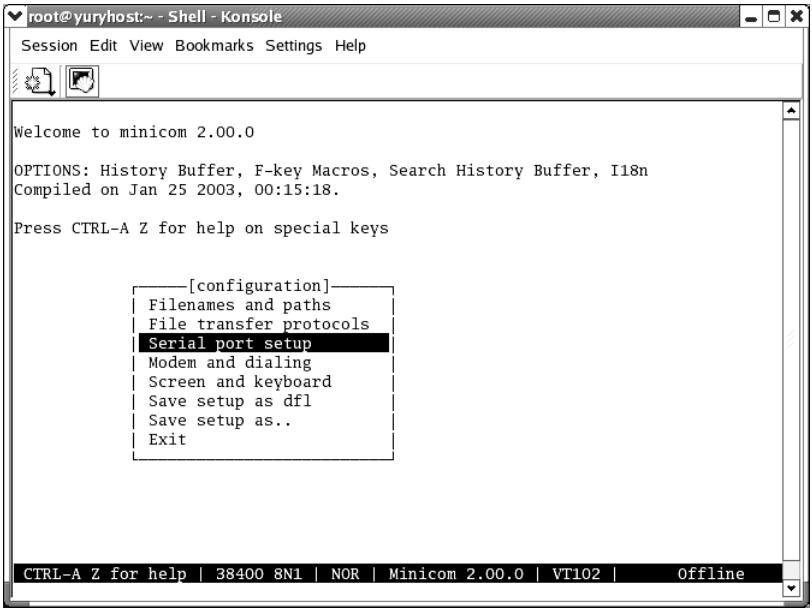


Рис. 5.5. Выбор опции настройки последовательного порта

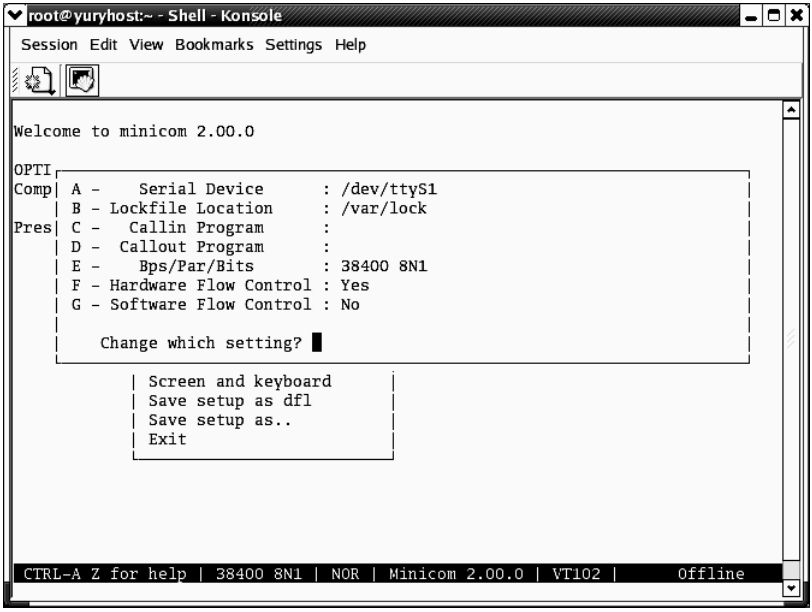


Рис. 5.6. Окно настройки устройства

После выбора опции настройки последовательного порта откроется следующее окно (рис. 5.6), в котором показаны опции для настройки параметров устройства.

Непосредственно для настройки последовательного порта используются следующие опции:

- A — выбор устройства (/dev/ttyS0, /dev/ttyS1 и т. д.);
- E — настройка скорости обмена (по умолчанию 38400 бод), четности (по умолчанию без контроля четности), количества бит посылки данных (по умолчанию 8) и количества стоповых битов (по умолчанию 1);
- F — установка аппаратного (метод RTS/CTS) управления потоком данных;
- G — установка программного (метод XON/XOFF) управления потоком данных.

В данном случае нам нужно установить скорость обмена данными, равную 9600 бод, поэтому выберем опцию E, нажав на клавиатуре соответствующую клавишу. В следующем окне (рис. 5.7) можно выбрать требуемые параметры.

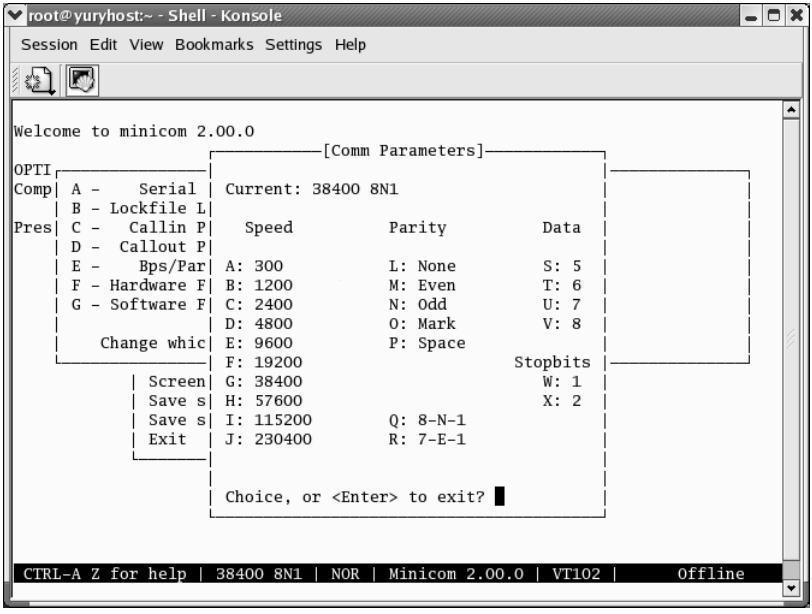


Рис. 5.7. Окно установки параметров обмена данными

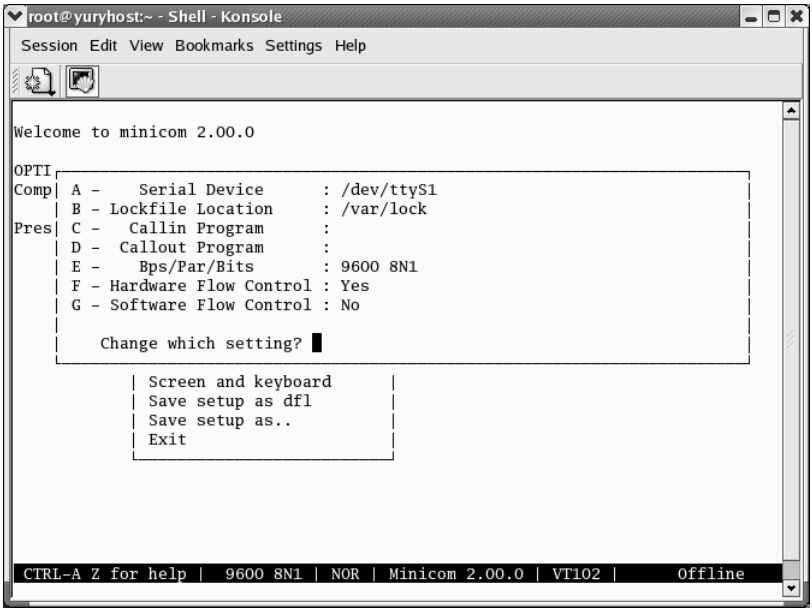


Рис. 5.8. Окно подтверждения выбора новых параметров

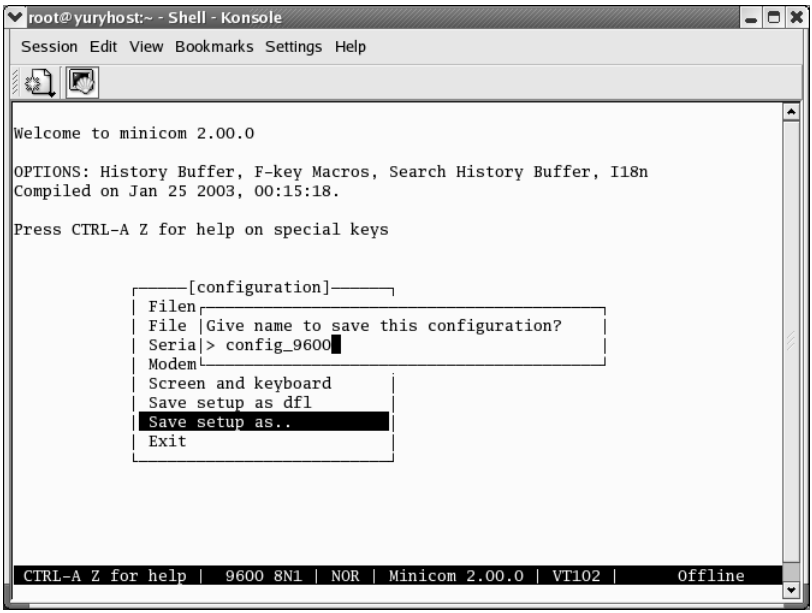


Рис. 5.9. Сохранение настроек в файле конфигурации

Из всех имеющихся опций в данном случае нам потребуется E. После нажатия клавиши <E> будет установлена скорость обмена 9600 бод. После нажатия клавиши <Enter> появится предложение сохранить сделанные изменения параметров (рис. 5.8).

После подтверждения изменений сохраняем новую конфигурацию в отдельном файле (рис. 5.9), который далее можно будет использовать в качестве параметра `configuration` при следующем запуске программы minicom.

В следующем разделе мы рассмотрим несколько примеров программирования последовательного порта на языке GNU C.

5.2. Примеры программирования последовательного порта

Для тестирования последовательного порта /dev/ttyS0 (COM1) используется внешнее устройство на базе микроконтроллера 8052 со встроенным последовательным портом, которое через нуль-модемный кабель подсоединено к последовательному порту ПК. Исходный текст программы, записанной во флэш-память программ этой системы, написан на ассемблере ASM51 и представлен в листинге 5.1.

Листинг 5.1. Тестовая программа внешней системы на базе микроконтроллера 8052

```
    NAME   PROCS
    MAIN   SEGMENT CODE
    CSEG   AT 0
    jmp     start
str:
   DB 'Test String For Linux', 0dh, 0ah, 1bh
   RSEGMAIN
start:
   mov      SCON, #50h
   mov      TH1, #0F3h
   orl      PCON, #80h
   mov      TMOD, #20h
   setb     TR1
   mov      DPTR, #str
```

```
again:
   clr      TI
   clr      A
   movc     A, @A+DPTR
   cjne     A, #1bh, write
   jmp      $
write:
   mov      SBUF, A
   jnb      TI, $
   inc      DPTR
   jmp      again
   end
```

Эта программа в момент запуска передает текстовую строку `str` в последовательный порт. Скорость обмена данными для системы выбрана 9600 бод, поэтому в тестируемой системе с операционной системой Linux следует установить такую же скорость обмена для последовательного порта.

Простейший способ тестирования последовательного порта в Linux можно выполнить из командной строки, используя механизм перенаправления ввода. В терминальном окне нужно ввести строку

```
# cat </dev/ttyS0
```

Эта команда перенаправляет данные, поступающие в первый последовательный порт системы Linux, команде `cat`, которая будет отображать текстовую строку всякий раз при запуске программы на микроконтроллере.

Ту же самую операцию — вывод строки байтов, полученной из последовательного порта, на экран консоли выполняет и программа, исходный текст которой показан в листинге 5.2.

Листинг 5.2. Чтение строки байтов из последовательного порта

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
```

```
int fd;
int bytesRead;
char buf[128];



fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NONBLOCK);
if (fd == -1)
 {
  printf("Cannot open port!\n");
  return 1;
  };

while (1)
 {
   bytesRead = read(fd, buf, sizeof(buf));
   if (bytesRead > 0)
   {
    buf[bytesRead] = '\0';
    printf("%s", buf);
   };
 };
 close(fd);
 return 0;
}
```

Для компиляции данной программы, исходный текст которой находится в файле readCOM.c, следует ввести командную строку

```
# g++ -o readCOM readCOM
```

Если компьютеры подключены к сети, то данные, полученные с последовательного порта одного компьтера, можно передать по сети другому компьютеру. Подобную систему мы уже анализировали применительно к операционным системам Windows. Внешнее устройство, данные с которого должны обрабатываться, присоединено к последовательному порту ПК. При поступлении строки данных через интерфейс RS-232 в компьютер под управлением Linux, приложение-клиент, запущенное на ПК, принимает ее и отправляет по сети TCP/IP приложению-серверу.

Для тестирования такой системы можно использовать одну и ту же машину, на которой запускается приложение-клиент и приложение-сервер. В этом случае обмен данными по сети осуществляется через IP-адрес обратной связи 127.0.0.1, который применяется для диагностических целей. Именно такая конфигурация и используется при тестировании передачи данных от последовательного порта компьютера в программе-клиенте TCP на программу-сервер. Оба приложения работают с портом 5151.

Исходный текст приложения-клиента показан в листинге 5.3.

Листинг 5.3. Программный код приложения-клиента

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>

#define port 5151

int main(void)
{

 char *addr = "127.0.0.1";
 int fd, fd_sock;
 struct sockaddr_in server;
 char buf[128];
 int bytesRead;

 fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NONBLOCK);
 if (fd == -1)
  {
   printf("Cannot open port!\n");
```

```
   return 1;
   };

bzero(&server, sizeof(server));

inet_aton(addr, &server.sin_addr);

server.sin_family = AF_INET;
server.sin_port = htons(port);

while (1)
{
 bytesRead = read(fd, buf, sizeof(buf));
 if (bytesRead > 0)
   {
    buf[bytesRead] = '\0';
    fd_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    connect(fd_sock, (struct sockaddr*)&server, sizeof(server));
    send(fd_sock, buf, bytesRead, 0);
    close(fd_sock);
   }
}
return 0;
}
```

В этой программе для чтения данных открывается последовательный порт /dev/tytyS0 (функция `read`). Чтение данных будет выполняться в неблокирующем режиме (опция `O_NONBLOCK`) в цикле `while (1)` основной программы. Передача данных серверу осуществляется через сокет `fd_sock`, который создается функцией `socket`. Данные передаются через порт 5151 на локальную машину с IP-адресом обратной связи 127.0.0.1.

Привязка номера порта и IP-адреса к сокету `fd_sock` выполняется с помощью функции `connect`:

```
connect(fd_sock, (struct sockaddr*)&server, sizeof(server));
```

Далее данные, предварительно полученные с последовательного порта /dev/ttyS0 функцией `read` и записанные в буфер `buf`, передаются по сети TCP-

серверу, который работает на одном и том же хосте, что и клиент. Передача содержимого буфера `buf` по сети осуществляется функцией `send`:

```
send(fd_sock, buf, bytesRead, 0);
```

По завершении передачи данных сокет закрывается функцией `close(fd_sock)`, и цикл повторяется.

Программа-сервер TCP принимает данные от клиента и отображает их на экране консоли. В данном случае сервер, как и клиент, работает на одном и том же хосте, используя для прослушивания IP-адрес обратной связи 127.0.0.1 и порт 5151. Если имеются свободные компьютеры в сети, то обе программы (клиент и сервер) можно запустить на разных машинах. В этом случае вместо IP-адреса 127.0.0.1 в приложении-клиенте следует указать конкретный адрес хоста, на котором работает TCP-сервер. Вместо номера порта 5151, который в данном примере выбран произвольно из имеющихся свободных портов, можно взять другой.

Исходный текст программы-сервера TCP показан в листинге 5.4.

Листинг 5.4. Программный код приложения-сервера TCP

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <arpa/inet.h>

#define port 5151

int main(void)
{

 char *addr = "127.0.0.1";
 int fd_sock, new_con;
 struct sockaddr_in server;
 char buf[128];
 int bytesRet;
```

```
bzero(&server, sizeof(server));

server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_family = AF_INET;
server.sin_port = htons(port);

fd_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
bind(fd_sock, (struct sockaddr*)&server, sizeof(server));
listen(fd_sock, 5);
printf("Waiting for data at port 5151...\n");
while(1)
{
 new_con = accept(fd_sock, 0, 0);
 if (new_con > 0)
  {
   bytesRet = recv(new_con, buf, sizeof(buf), 0);
   if (bytesRet > 0)
    {
     buf[bytesRet] = '\0';
     printf("%s\n", buf);
    }
   close(new_con);
   }
 }
close(fd_sock);
return 0;
}
```

Программа-сервер TCP работает в блокирующем режиме, прослушивая запросы на соединение на порту 5151. Прослушивающий сокет `fd_sock` создается со всеми необходимыми привязками с помощью функций `socket` и `bind`.

Если привязка выполнена успешно, то сокет запускается в режим прослушивания функцией `listen`. Инициализация новых клиентских соединений выполняется в бесконечном цикле `while (1)` функцией `accept`. При создании нового соединения с сокетом `new_con` сервер принимает данные от клиента с помощью функции `recv` в буфер памяти `buf`:

```
bytesRet = recv(new_con, buf, sizeof(buf), 0);
```

Если количество принятых байтов (переменная `bytesRet`) отлично от нуля, то содержимое буфера выводится на консоль. После выполнения функции `recv` рабочий сокет `new_con` закрывается.

Для проверки работы сервера и клиента TCP следует запустить обе программы в разных терминальных окнах.

Следующий пример программы, который мы рассмотрим, демонстрирует работу приложения в режиме демона. Демоны операционных систем UNIX представляют собой, как и системные сервисы Windows, особый класс приложений. Они предназначены для выполнения задач при отсутствии управляющего терминала программы и являются, в определенном смысле, изолированными приложениями. Для создания демона в основном процессе программы следует вызвать функцию `fork`, с помощью которой создается новый процесс.

Затем следует завершить основной процесс, например, командой `exit`, а в порожденном процессе оборвать связь с управляющим терминалом программы с помощью функции `setsid`. После этого порожденный процесс становится лидером группы сеансов и выполняется как самостоятельный процесс. Демон будет читать данные с последовательного порта /dev/ttyS0 и выводить их на экран консоли. Строго говоря, приложения-демоны, в общем случае, теряют связь с консолью запущенной основной программы и не могут выводить в нее данные, если только специально не запрограммировать эту возможность. В данном упрощенном примере, пока пользователь не вышел из системы, данные будут выводиться на экран консоли.

Исходный текст программы-демона (файл Daemon1.c) показан в листинге 5.5.

Листинг 5.5. Программный код приложения-демона

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>

int main (void)
{
 pid_t pid;
 int fd;
```

```
int bytesRead;
char buf[128];

fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NONBLOCK);
if (fd == -1)
 {
  printf("Cannot open port!\n");
  return 1;
  };

chdir("/root");
pid = fork();
if (pid > 0)
 {
  close(fd);
  exit(0);
 }
setsid();
printf("Daemon works...\n");
while (1)
{
 bytesRead = read(fd, buf, sizeof(buf));
   if (bytesRead > 0)
   {
    buf[bytesRead] = '\0';
    printf("%s", buf);
   };
 }
 close(fd);
 return 0;
}
```

В этой программе с помощью функции `open` открывается последовательный порт /dev/ttyS0 для чтения-записи в неблокирующем режиме. Затем с помощью функции `fork` создается новый процесс:

```
pid = fork();
```

Идентификатор процесса помещается в переменную `pid`. Особенностью функции `fork` является то, что она возвращает разные идентификаторы порождающему (основному) и порожденному процессам. В основной процесс возвращается опеределенное значение, отличное от нуля, а в порожденный процесс возвращается 0. Это позволяет разделить программный код основного и порожденного процесса. Поскольку мы создаем демона, то основной процесс следует завершить, что выполняют операторы

```
if (pid > 0)
  {
   close(fd);
   exit(0);
  }
```

Поскольку порожденный процесс наследует дескрипторы открытых файлов основного процесса, то дескриптор `fd` в основном процессе нам больше не нужен. Его можно закрыть функцией `close`, после чего завершить основной процесс функцией `exit`.

Далее выполняется только порожденный процесс. Функция `setsid` обрывает связь процесса с терминалом порождающего процесса и делает порожденный процесс лидером группы сеансов. Процесс-демон будет выполнять чтение данных с последовательного порта в бесконечном цикле `while (1)` с помощью функции `read`. Программа-демон компилируется и запускается обычным образом:

```
# g++ -o Daemon1 Daemon1.c
# ./Daemon1
```

Завершить процесс-демон можно только командой `kill` операционной системы. Для этого следует определить идентификатор процесса с помощью команды `ps`:

```
[root@yuryhost root]# ps -ef|grep Daemon1
root      5586     1 96 22:23 ?        00:00:22 ./Daemon1
root      5616  5588  0 22:24 pts/2    00:00:00 grep Daemon1
```

После этого можно ввести команду `kill` с соответствующими параметрами:

```
# kill 9 5586
```

Первый параметр этой команды, равный 9, означает безусловное уничтожение процесса, а второй равен значению идентификатора процесса.

В следующем, более сложном проекте, рассмотрим операции записи и чтения байтов через последовательный порт. Для этого в память программ системы на микроконтроллере запишем программный код на языке Keil C51 (листинг 5.6).

Листинг 5.6. Тестовая программа для внешнего устройства на базе микроконтроллера 8052

```
#include <REG52.H>
#include <stdio.h>

void main(void)
{
 char c1;
 unsigned int cnt = 0;

 SCON = 0x50;
 TMOD |= 0x20;
 TH1 = 0xF3;   //скорость 9600 бод при Fosc = 24,0 МГц
 PCON |= 0x80;
 TR1 = 1;
 TI = 1;

 c1 = getkey();
 printf("String returned.\n");
 while (1);
  }
```

Эта программа ожидает поступления байта с последовательного интерфейса, после чего отправляет в последовательный порт строку байтов. Для этого приложения выбрана стандартная скорость обмена данными 9600 бод, поэтому в тестируемой системе на базе Linux следует установить такие же параметры скорости обмена для последовательного порта.

С помощью этого приложения будет продемонстрировано выполнение операций чтения-записи в операционной системе Linux. Напомню, что для демонстрации выполнения операций чтения-записи внешнее устройство с микроконтроллером должно быть подключено к первому последовательному порту ПК.

Для проверки работы последовательного порта в Linux можно в двух разных терминальных окнах набрать две команды. Сначала набирается команда

```
# cat < /dev/ttyS0
```

а затем

```
# echo 1 > /dev/ttyS0
```

При выполнении команды с `echo` команда с `cat` прочитает данные с первого последовательного порта /dev/ttyS0, отобразив строку

```
String returned.
```

Таким образом, мы проверили работу последовательного порта в Linux с помощью интерпретатора командной строки и команд перенаправления потока данных. Для подобной проверки с помощью отдельной программы скомпилируем исходный текст на языке C, показанный в листинге 5.7.

Листинг 5.7. Тестовое приложение для проверки операций чтения-записи через последовательный порт

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
 int fd;
 int bytesRead;
 char buf[128];
 pid_t pid;
 char c1;

 fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NONBLOCK);
 if (fd == -1)
  {
   printf("Cannot open port!\n");
   return 1;
   };
 pid = fork();
 if (pid == 0)
 {
  while (1)
   {
    bytesRead = read(fd, buf, sizeof(buf));
```

```
    if (bytesRead > 0)
    {
     buf[bytesRead] = '\0';
     printf("%s", buf);
    };
  };
  close(fd);
 }
 else
  {
   while (1)
    {
    c1 = getchar();
    write(fd, &c1, 1);
    }
  }
   close(fd);
  return 0;
 }
```

В нашем приложении для обмена данными используется файл устройства /dev/ttyS0, открытый в неблокирующем режиме функцией `open`.

Для одновременного чтения и записи данных в программе используется два отдельных процесса. Основная программа записывает полученный с клавиатуры консоли байт данных в последовательный порт /dev/ttyS0 с помощью функции `write` в цикле `while (1)`:

```
while (1)
    {
    c1 = getchar();
    write(fd, &c1, 1);
    }
```

Для чтения данных с последовательного порта с помощью функции `fork` запускается отдельный процесс, в котором чтение данных выполняется в цикле `while (1)`:

```
while (1)
   {
```

```
    bytesRead = read(fd, buf, sizeof(buf));
    if (bytesRead > 0)
    {
     buf[bytesRead] = '\0';
     printf("%s", buf);
    };
  };
```

Здесь функция `read` читает данные с порта /dev/ttyS0 в буфер `buf`, после чего, если получены какие-то данные (значение переменной `bytesRead` > 0), происходит их отображение на экране консоли.

Сохраним исходный текст программы в файле readCOM.c и откомпилируем его с помощью командной строки

```
# g++ -o readCOM readCOM.c
```

После компиляции исходного текста можно запустить программу readCOM и проверить, как выполняется чтение-запись через устройство /dev/ttyS0. Для этого на клавиатуре вводится любой символ и наблюдается результат, как показано далее:

```
# ./readCOM
d
String returned.
```

Можно проверить последовательный порт и другим методом. Для этого следует в одном терминальном окне запустить программу readCOM, а в другом ввести команду

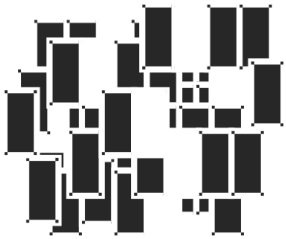
```
# echo 1 > /dev/ttyS0
```

и наблюдать результат.

Для программирования последовательного порта в операционной системе Linux, кроме языка C, можно воспользоваться и другими инструментальными средствами, например, интерпретатором Perl или Java. В частности, для интерпретатора Perl разработано несколько модулей, позволяющих очень легко запрограммировать операции обмена данными через последовательный порт.

Дополнительную информацию по программированию последовательного интерфейса в операционных системах Linux можно обнаружить на многочисленных форумах, посвященных этой теме.

Глава 6



Расширения ввода-вывода последовательного интерфейса

В этой главе рассматриваются вопросы взаимодействия оборудования с интерфейсом RS-232 с другими аппаратно-программными интерфейсами современных компьютерных систем, такими как USB и TCP/IP. Многие компьютерные системы и внешние устройства в настоящее время выпускаются только с одним интерфейсом USB (Universal Serial Bus), в то время как программное обеспечение и внешнее оборудование допускают работу с интерфейсом RS-232. К такому оборудованию относится большинство систем на базе микроконтроллеров и микропроцессоров, выполняющих первичную обработку данных и функции управления в промышленности, науке и медицине. В наладке этих систем очень часто используются компьютеры, например, ноутбуки, не имеющие встроенного COM-порта и, естественно, возможности прямого подключения. Проблема легко решается, если использовать аппаратно-программные устройства, преобразующие протокол RS-232 в USB.

Аппаратно такие устройства включают кристалл специализированной микросхемы, например, FTDI, Cypress или др., который преобразует стек одного протокола в стек другого. Перед тем как использовать преобразователь RS-232 в USB, следует внимательно ознакомиться с технической документацией на данное устройство. Нужно учитывать и то, что протокол RS-232 определяет параметры обмена данными на уровне сигналов интерфейса, но не на уровне приложения. В то же время протокол USB позволяет работать на более высоком уровне абстракции. В этом случае программное взаимодействие интерфейсов RS-232 и USB целиком возлагается на разработчика программного обеспечения.

Установка параметров интерфейса RS-232 (скорости обмена, контроля четности и т. д.) относительно легко выполняется из приложения пользователя.

USB-интерфейс такой возможности не дает, поэтому для достижения нужных характеристик обмена данными при преобразовании RS-232 в USB нужен, как правило, дополнительный драйвер (кроме имеющихся в системе), который будет эмулировать функции асинхронного приемопередатчика (UART) интерфейса RS-232, но взаимодействовать с системой фактически по USB.

Часто при преобразовании RS-232 в USB возникают проблемы аппаратной совместимости по уровням сигналов. Аппаратная часть интерфейса RS-232 персонального компьютера использует питающие напряжения системы +5 В для UART и ±12 В для преобразователей уровня выходных буферов, таких, например, как MAX232. Для питания USB нужен только один источник с напряжением +5 В. Некоторые преобразователи RS-232 в USB имеют интегрированный преобразователь постоянного напряжения, чтобы обеспечить такие уровни сигналов. Тем не менее, в недорогих преобразователях для питания всей схемы, включая выходные каскады буферных усилителей, используется напряжение +5 В. Сигналы интерфейса RS-232 с такими уровнями теоретически попадают в допустимый диапазон, однако такие заниженные уровни могут привести к попаданию сигнала в зону неопределенности от –3 В до +3 В, что влечет за собой сбои в работе оборудования. Широко известные микросхемы буферных усилителей-преобразователей MAX232 ведут себя именно таким образом.

Еще одной серьезной аппаратной проблемой при использовании преобразователей является переполнение буфера на стороне приемника. Для управления потоком данных и предотвращения подобных ситуаций применяется аппаратное (RTS/CTS) или программное (XON/XOFF) управление потоком данных. Следует иметь в виду, что не во всех преобразователях RS-232 в USB реализована такая возможность. При возникновении таких проблем аппаратура может работать хорошо, но будут происходить потери данных в приложении из-за переполнения буфера приема. В целом ряде случаев, например, если приложение не использует управление потоком данных, можно не беспокоиться об отсутствии такой возможности. Но при большой загрузке процессора или при получении больших объемов данных проблема управления потоком может стать актуальной.

При выборе преобразователя RS-232 в USB нужно тщательно проанализировать временные зависимости обмена данными. Если имеются жесткие ограничения на время приема и обработки данных, то лучше отказаться от выбора внешнего преобразователя в пользу внутреннего UART — дополнительное звено в цепи обработки данных в виде вспомогательного драйвера только замедлит работу всей системы.

6.1. Виртуальные порты

В настоящее время очень широкое распространение среди разработчиков программного обеспечения для коммуникационных систем получил метод тестирования и диагностики программного обеспечения с использованием так называемых *виртуальных устройств*. Виртуальные устройства моделируют поведение реальных систем на программном уровне, позволяя разработчику выполнить отладку приложений до того, как будет создан реальный физический прототип устройства. Этот метод широко применяется при разработке самых сложных коммуникационных систем, включая и устройства с интерфейсами RS-232/RS-485. В настоящее время выпускается широкий спектр программных средств, позволяющих создавать в операционной системе (чаще всего в Windows) несколько виртуальных последовательных портов, которые разработчик может использовать для тестирования обмена данными по интерфейсам RS-232/RS-485.

Серьезным преимуществом применения виртуальных последовательных интерфейсов является и то, что они позволяют тестировать и отлаживать программное обеспечение при отсутствии физического последовательного порта. Например, для полноценной отладки программ обмена данными по последовательному порту требуется наличие как минимум двух физических портов. Поскольку современные ПК выпускаются с одним последовательным портом, то для отладки приложения требуется два компьютера или же дополнительная PCI-плата расширения с последовательным портом. В то же время на одном и том же ПК программно можно создать несколько виртуальных последовательных портов, которые позволят выполнить полноценное тестирование программы обмена данными. Второй очень важный момент, о котором следует упомянуть, заключается в том, что концепция виртуальных портов может быть использована (и широко применяется) не только для разработки коммуникационных приложений, но и в программировании обычных приложений.

Виртуальные порты с успехом могут применяться для обмена данными между двумя и более приложениями без использования довольно сложной программной технологии межпроцессных коммуникаций и разделения ресурсов. Поскольку здесь используются виртуальные, а не реальные физические порты, то скорость обмена данными между приложениями будет определяться не характеристиками порта, а параметрами системы (типом процессора, объемом доступной физической памяти и т. д.).

Для обмена данными по виртуальным последовательным портам в операционных системах Windows используются стандартные функции для работы с файлами и устройствами (`CreateFile`, `ReadFile`, `WriteFile` и `CloseHandle`). Кроме того, для программирования виртуальных портов можно использовать и библиотечные функции C++, такие как `open`, `read`, `write` и `close`, а при разработке программ в среде Microsoft Visual Studio. NET можно применить компонент SerialPort.

Для установки в систему виртуальных последовательных портов существует множество программ различных фирм-производителей программного обеспечения, среди которых наиболее мощной в плане функциональности является очень популярная программа фирмы Eltima Software GmbH под названием *Virtual Serial Port Driver* (VSPD), работающая в операционных системах линейки Windows. Далее мы детально ознакомимся с принципами функционирования этого приложения, тем более что механизмы, положенные в ее основу, используются и в других подобных приложениях других фирм-производителей.

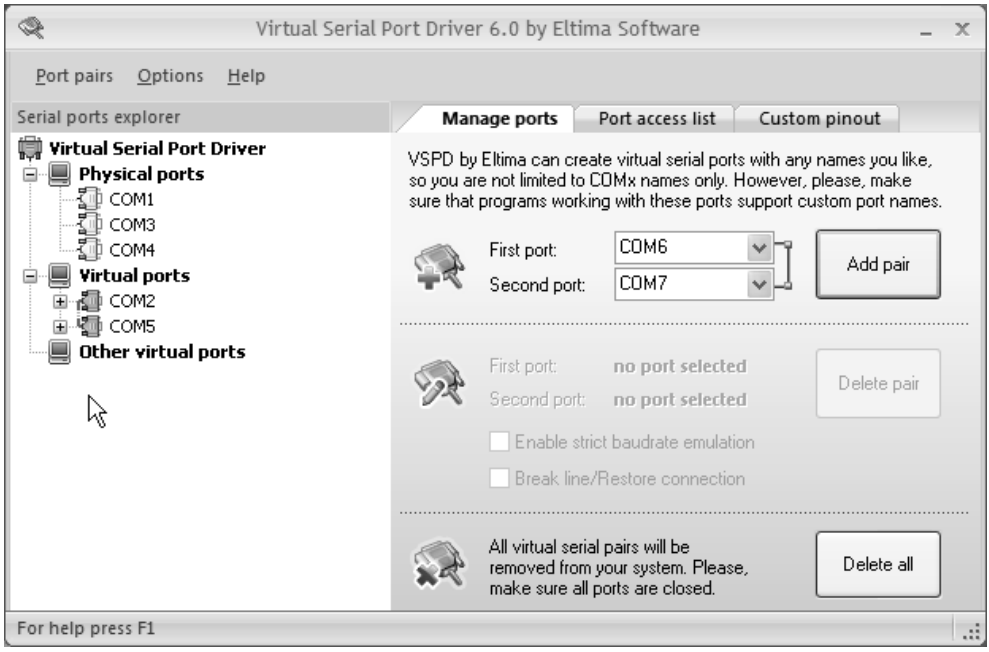


Рис. 6.1. Установка виртуальных последовательных портов

Полнофункциональная версия программы-инсталлятора виртуальных последовательных портов стоит недорого, и, кроме того, для ознакомления всегда можно закачать последнюю демо-версию программы с сайта **www.eltima.com**.

При анализе особенностей функционирования программы VSPD мы будем использовать версию 6, хотя все рассуждения можно отнести и к более ранним версиям приложения. После установки приложения в систему можно добавить виртуальные порты, как показано на рис. 6.1.

Для установки виртуальных портов нужно нажать кнопку **Add pair**, при этом программа установит пару виртуальных портов, которые следуют за физическими портами, имеющимися в системе. В данной конфигурации порт COM1 зарезервирован системой, порты COM3 и COM4 используются платой расширения интерфейса, поэтому программа VSPD устанавливает первую пару портов с номерами 2 и 5.

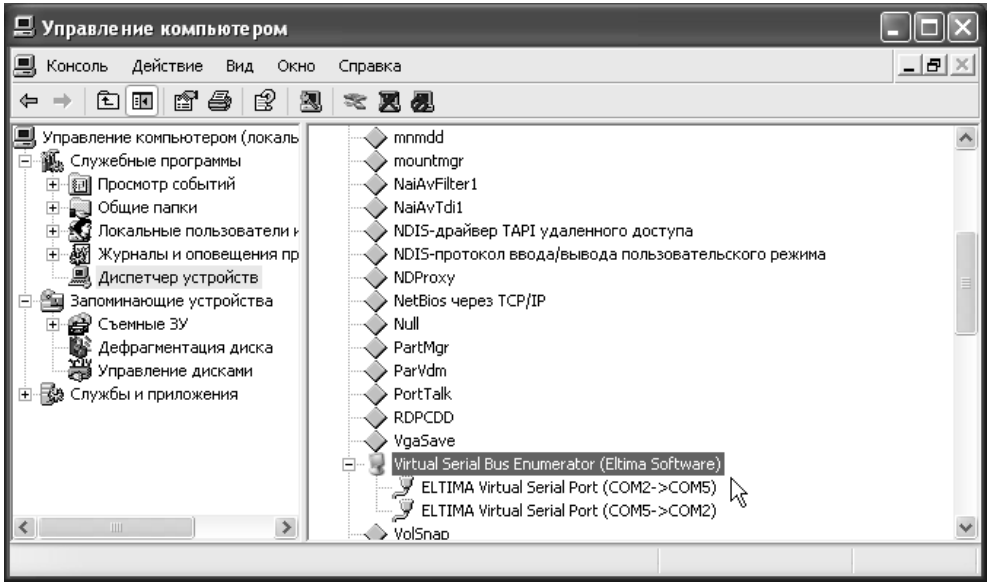


Рис. 6.2. Пример конфигурации виртуальных портов

Виртуальные порты COM2 и COM5 соединяются таким же виртуальным нуль-модемным соединением, поэтому их можно использовать для обмена данными между различными приложениями, использующими интерфейс RS-232. Сконфигурированные таким образом последовательные порты по

умолчанию не используют установки скорости обмена данными (опция **Baudrate emulation** установлена в **Disabled**). В этом случае обмен данными по виртуальным портам будет определяться установками соответствующих параметров в приложениях, использующих эти порты.

Программная архитектура приложения VSPD, как и других подобных программ-"виртуализаторов", включает два виртуальных устройства с каналом передачи данных. Конфигурация такой системы показана на рис. 6.2.

На этом рисунке показана возможная конфигурация виртуальных устройств. Перечислитель шины устанавливает два устройства COM2 и COM5 в систему. Эти устройства в пространстве имен (Object Name Space) операционной системы имеют имена VSerial_0 VSerial_1 и управляются драйвером evserial.sys (рис. 6.3).

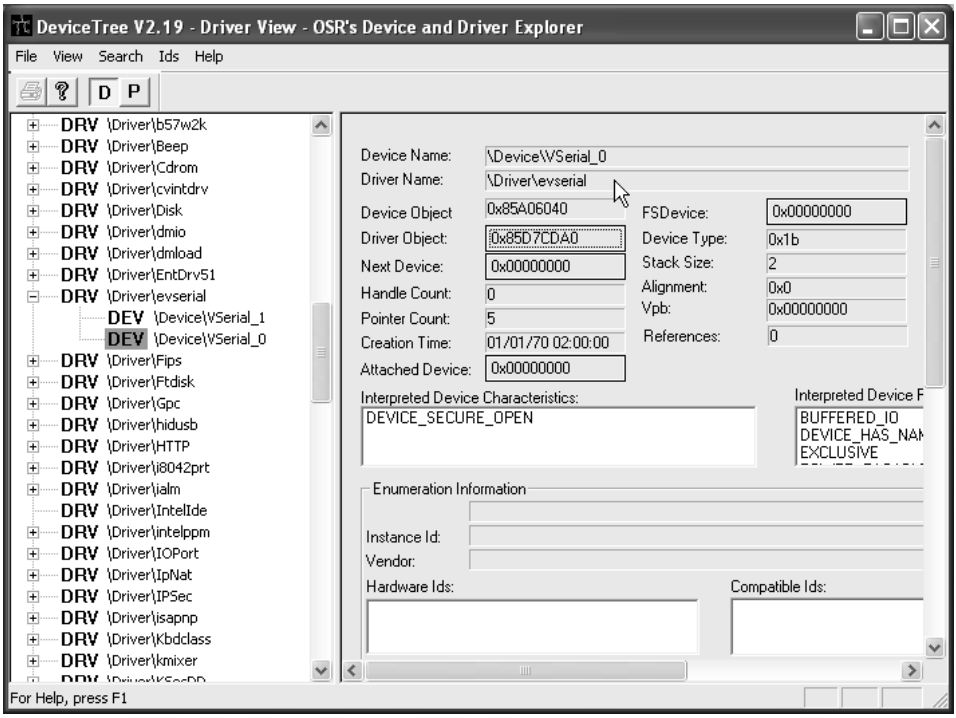


Рис. 6.3. Взаимосвязь драйвера устройства и виртуальных портов

Драйвер виртуальных устройств программно реализует протокол RS-232, включая настройки программного (XON/XOFF) и аппаратного (RTS/CTS)

управления потоком данных. Программа, выполняющая обмен данными через виртуальный порт COM2 или COM5, может использовать возможности аппаратно-программного управления потоком данных установкой соответствующих параметров в функциях `SetCommState`, `EscapeCommFunction` и др. Единственным ограничением виртуального порта является невозможность работы с физическими портами ввода-вывода, поскольку таковые здесь отсутствуют, но для большинства Windows-приложений это и не требуется.

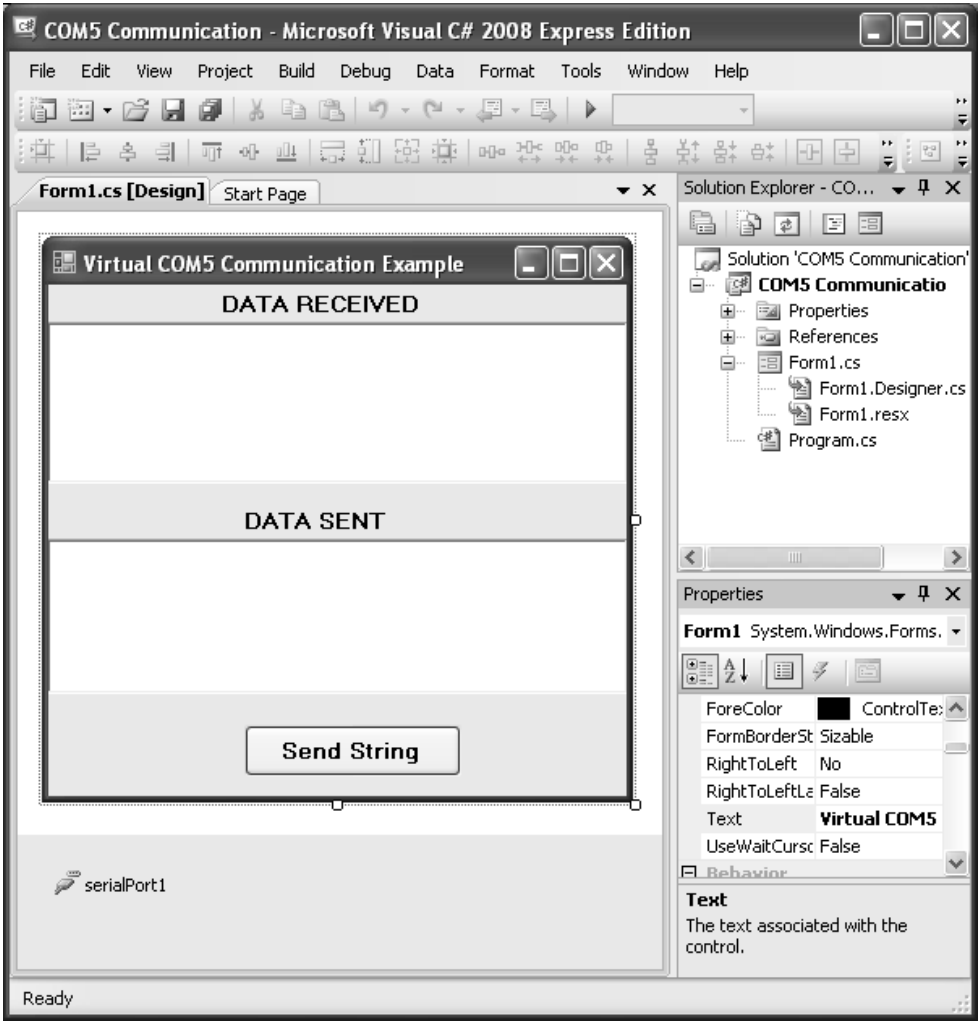


Рис. 6.4. Окно конструктора приложения

Для демонстрации возможностей разработки и отладки приложений с использованием виртуальных портов рассмотрим несколько программных проектов. В них будут задействованы два приложения: программа-эмулятор терминала, работающая на виртуальном порту COM2, и наше тестовое приложение, выполняющее обработку данных на виртуальном порту COM5. Тестовое приложение разработаем в среде Visual C# .NET 2008 Express Edition. Окно конструктора приложения с размещенными на нем компонентами показано на рис. 6.4.

В этом приложении на форме размещаются два многострочных текстовых редактора RichTextBox. В окно одного из них будет выводиться текстовая строка, полученная с виртуального последовательного порта COM2, а в окно второго будет вводиться текст для отправки в последовательный порт COM2. На форме приложения размещается кнопка **Send String**. При нажатии этой кнопки вызывается соответствующая функция-обработчик, посредством которой текстовая строка передается в последовательный порт.

Для обработки данных с виртуального порта COM2 в нижней части окна конструктора размещается экземпляр компонента SerialPort. Для этого компонента нужно установить номер порта, равный 5, скорость обмена 9600 бод, режим без проверки на четность и один стоповый бит. Остальные параметры можно оставить без изменения. Такие же параметры должны быть установлены и для программы-эмулятора терминала.

Исходный текст приложения приведен в листинге 6.1.

Листинг 6.1. Программа обмена данными через виртуальный порт COM5

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO.Ports;
using System.Windows.Forms;
```

```
namespace SendString_COM4
{
  public partial class Form1 : Form
  {
   public Thread myThread;
   public Form1()
   {
    InitializeComponent();
   }

  private void Form1_Load(object sender, EventArgs e)
  {
   serialPort1.Open();
   myThread = new Thread(ThreadProc);
   myThread.Start();
  }

  private void Form1_FormClosing(object sender,
                      FormClosingEventArgs e)
   {
    myThread.Abort();
    serialPort1.Close();
   }
   void ThreadProc()
   {
    this.serialPort1.DataReceived += new Sys-
tem.IO.Ports.SerialDataReceivedEventHandler
(this.serialPort1_DataReceived_1);
   while (true);
   }
  private void serialPort1_DataReceived_1
(object sender, SerialDataReceivedEventArgs e)
  {
   String sin = serialPort1.ReadLine();
   richTextBox1.AppendText(sin);
  }
```

```
  private void sButton_Click(object sender, EventArgs e)
   {
     String sout;
     sout = richTextBox2.Text;
     serialPort1.WriteLine(sout);
   }
 }
}
```

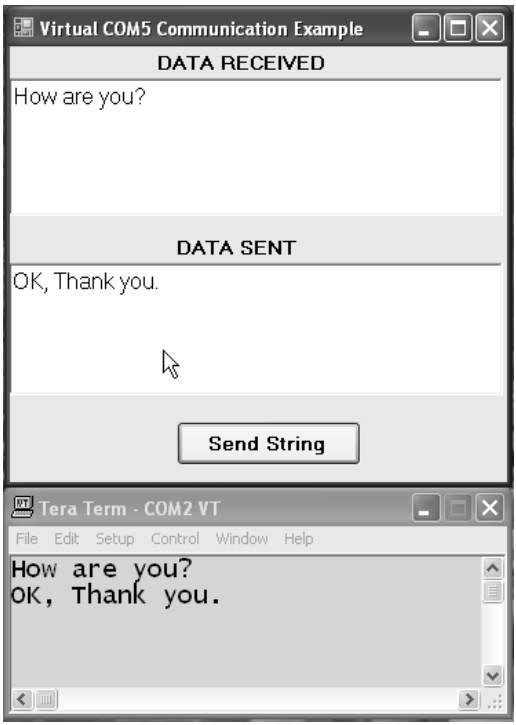


Рис. 6.5. Окно работающего тестового приложения

Основную работу в этом приложении выполняет компонент serialPort1. Для приема данных с последовательного порта и отображения их в окне текстового редактора в программе устанавливается функция-обработчик `serialPort1_DataReceived` события `DataReceived`, которая должна запускаться в отдельном потоке. Для этой цели создается поток `myThread` с функцией потока `ThreadProc`, в которой запускается функция `serialPort1_DataReceived`.

Передача строки данных через виртуальный последовательный порт COM5 выполняется в функции-обработчике `sButton_Click` кнопки `sButton`.

Для тестирования работы приложения запустим на виртуальном порту COM2 программу-эмулятор терминала, а на виртуальном порту COM5 — наше демонстрационное приложение. Данные, полученные от терминальной программы, отображаются в окне многострочного редактора с заголовком DATA RECEIVED. Для передачи данных программе-эмулятору в окне DATA SENT набирается текстовая строка, после чего следует нажать кнопку **Send String**. Окно нашего работающего приложения показано на рис. 6.5.

Поток данных через виртуальные последовательные порты COM2 и COM5 можно проконтролировать, открыв окно программы VSPD (рис. 6.6).

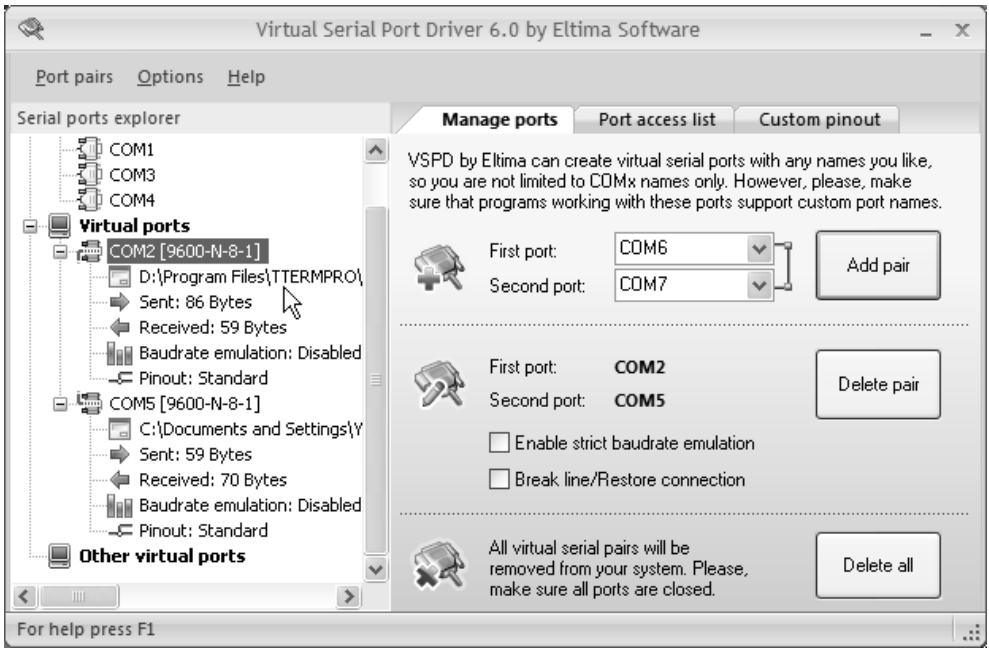


Рис. 6.6. Окно конфигурации и состояния программы VSPD

Из рисунка видно, какие параметры обмена данными установлены для виртуальных портов COM2 и COM5, а также приводится информация об использовании портов приложениями. О работоспособности системы можно косвенно судить по количеству переданных или принятых байтов данных через оба порта.

Виртуальные COM-порты, созданные программой VSPD, допускают программное (метод XON/XOFF) или аппаратное (метод RTS/CTS) управление потоком данных. Предыдущий пример несложно модифицировать для управления потоком данных. Для программного управления потоком данных на форме приложения разместим еще две кнопки (рис. 6.7).

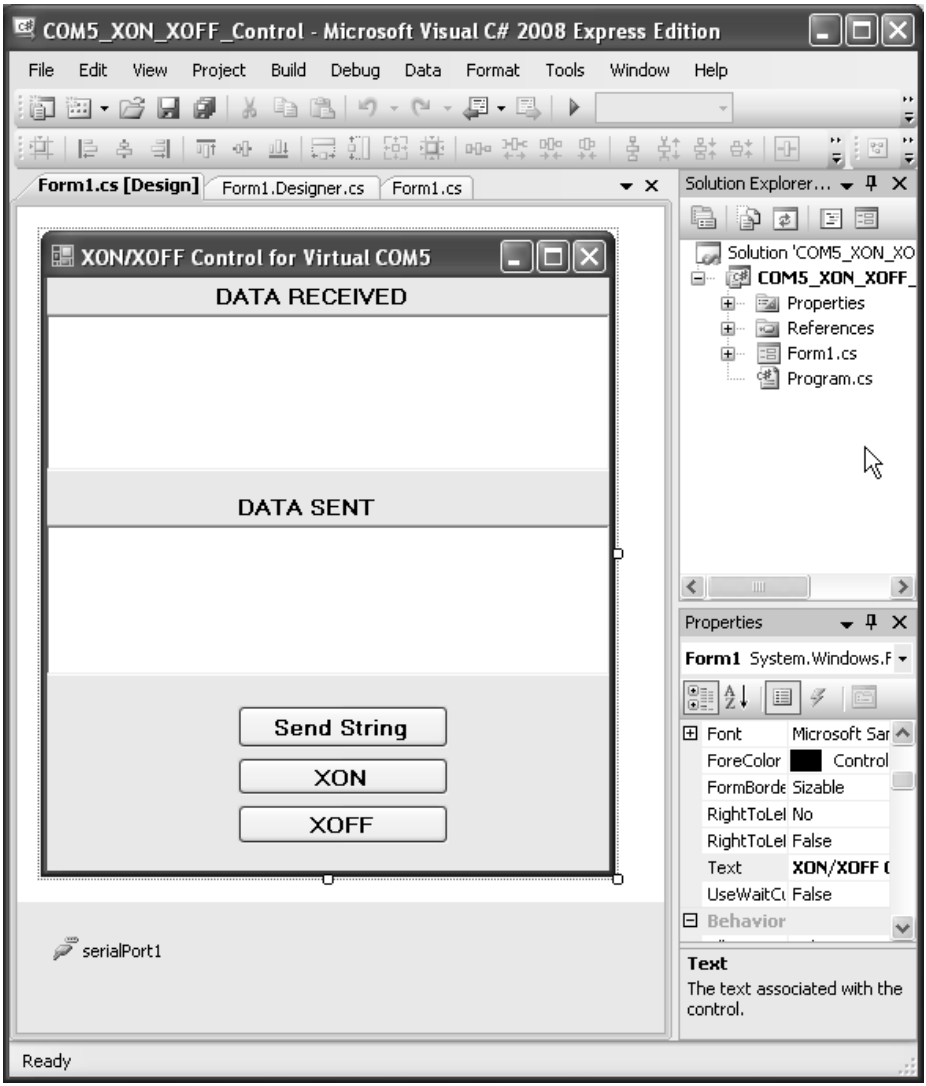


Рис. 6.7. Окно конструктора приложения

При нажатии кнопки с заголовком **XON** терминальной программе, работающей на виртуальном порту COM2, посылается байт XON (17), который разрешает передачу данных. Если нажата кнопка **XOFF**, то передается байт XOFF (19), запрещающий передачу данных. В программе-эмуляторе терминала обязательно должна быть установлена опция, разрешающая программный (XON/XOFF) контроль над потоком данных.

Основная часть исходного текста приложения позаимствована из предыдущего примера, добавлены только функции-обработчики нажатия кнопок **XON** и **XOFF**. Исходный текст функций-обработчиков приведен в листинге 6.2.

Листинг 6.2. Функции-обработчики событий XON и XOFF

```
private void xonButton_Click(object sender, EventArgs e)
    {
     byte [] XON = new byte [1];
     XON[0] = 17;
     serialPort1.Write(XON, 0, 1);
    }

private void xoffButton_Click(object sender, EventArgs e)
    {
     byte[] XOFF = new byte[1];
     XOFF[0] = 19;
     serialPort1.Write(XOFF, 0, 1);
    }
```

В обработчиках используется стандартный код байтов XON и XOFF, который передается в виртуальный порт COM5 посредством метода `Write`.

Еще один метод управления потоком данных — аппаратный (RTS/CTS) — реализован в приложении, окно конструктора которого показано на рис. 6.8.

В этом приложении при нажатии кнопки **RTS = 1** разрешается передача данных программе-терминалу, а при нажатии кнопки **RTS = 0** — запрещается. Программа-эмулятор терминала должна иметь установленную опцию, разрешающую RTS/CTS управление потоком.

Далее приводится исходный текст функций-обработчиков нажатий кнопок **RTS = 1** и **RTS = 0** (листинг 6.3).

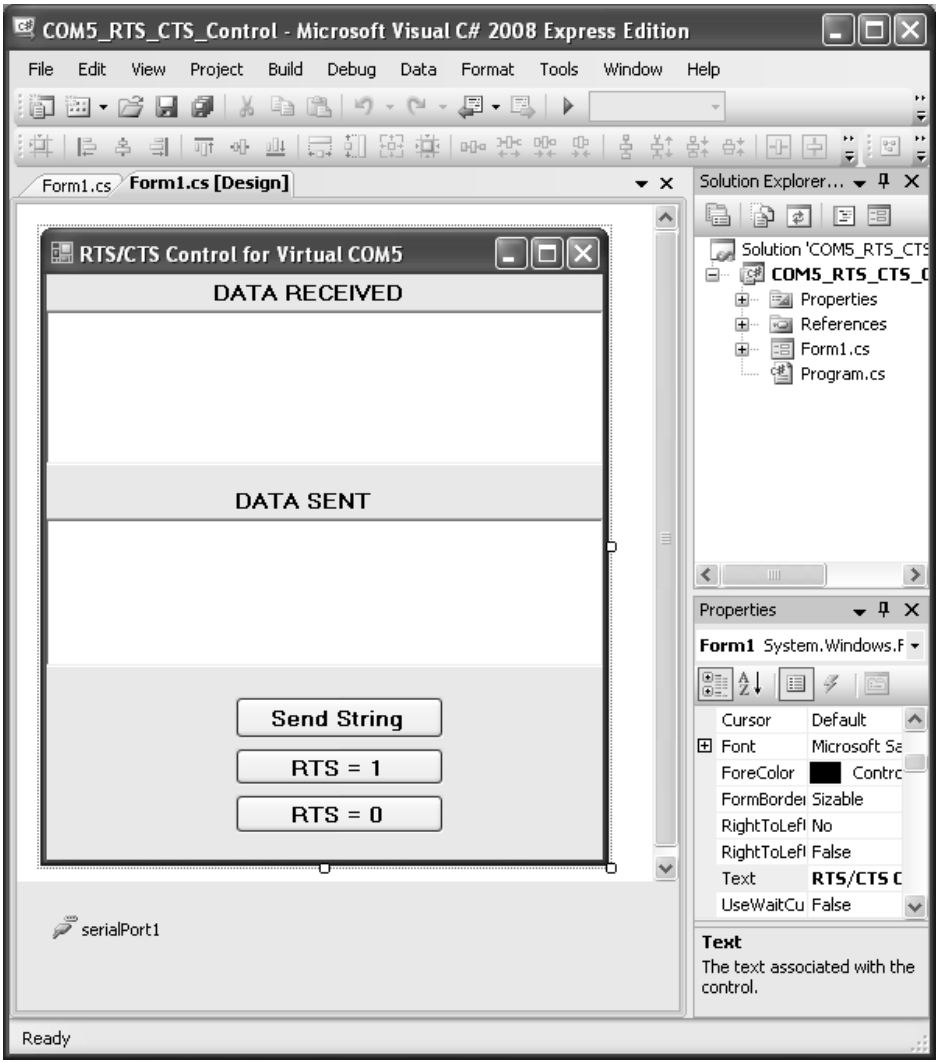


Рис. 6.8. Окно конструктора приложения

Листинг 6.3. Функции-обработчики событий

```
private void setButton_Click(object sender, EventArgs e)
    {
     serialPort1.RtsEnable = true;
    }
```

```
private void clrButton_Click(object sender, EventArgs e)
    {
     serialPort1.RtsEnable = false;
    }
```

Для разрешения передачи данных свойство `RtsEnable` компонента serialPort1 должно быть установлено в True. Чтобы запретить передачу данных, это же свойство должно иметь значение False.

6.2. Виртуальный интерфейс RS-232 и сеть Интернет

При обмене данными по интерфейсам RS-232 и RS-485 во многих случаях требуется передавать информацию на большие расстояния. Удаленная компьютерная система может находиться в десятках и сотнях километров от объекта, с которого передаются или принимаются данные. Наилучшим и самым доступным решением при выборе конфигурации такой распределенной системы является использование сети Интернет (рис. 6.9).

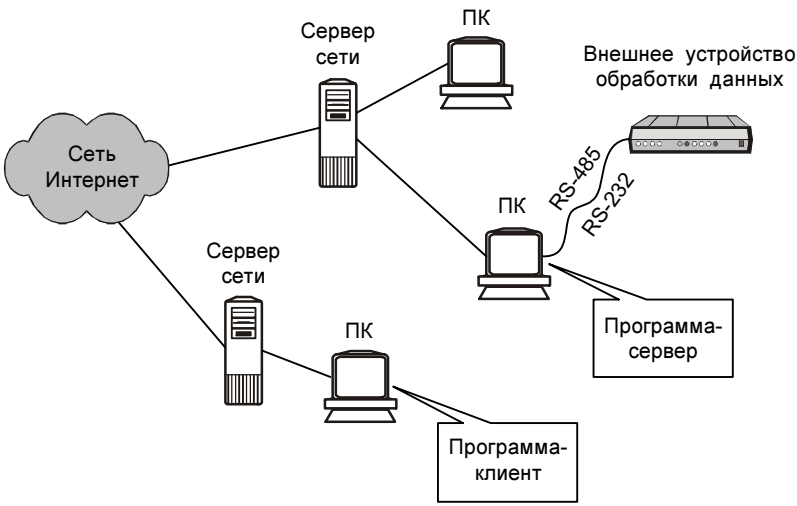


Рис. 6.9. Интеграция системы с RS-232/RS-485 в сеть Интернет

Для реализации такой системы на практике понадобится как минимум два компьютера, подключенных к сети Интернет. На одном (сервер) будет выполняться обработка данных от внешней системы, подключенной по интерфейсу RS-232/RS-485, а другой (клиент) будет считывать полученные данные по сети Интернет. Тестирование и проверка такой системы в реальных условиях потребует подключения и настройки (аппаратной и программной) внешней измерительной системы по RS-232/RS-485 и конфигурирования сетевых настроек персональных компьютеров.

Тем не менее, используя современные средства разработки, выполнить большую часть программной настройки такой системы обработки данных можно даже на одном персональном компьютере. Для этого не потребуется даже внешнее устройство, подключенное по интерфейсу RS-232/RS-485 к одному из компьютеров, поскольку функции передачи данных по последовательному интерфейсу можно целиком смоделировать с помощью виртуального последовательного порта. Что же касается сетевой части системы, то здесь также не возникает проблем — разработку и проверку приложения-клиента сети Интернет и Web-сервера можно выполнить на одной машине, задействовав интерфейс обратной связи с IP-адресом 127.0.0.1.

Конечно, окончательная доводка и отладка программной системы потребует установки полного комплекта оборудования и настройки сети Интернет, но время разработки всей системы значительно сократится одновременно с повышением качества ее работы.

Реализацию задачи разработки несложной системы обработки данных с последующей их передачей по сети Интернет нужно начать с конфигурирования последовательного интерфейса передачи данных между внешней системой сбора и обработки данных и персональным компьютером. На практике это можно осуществить несколькими способами:

- использовать полнофункциональное аппаратно-программное решение на базе промышленных протоколов Modbus, Profibus и т. д. Такое решение при его хорошей эффективности является дорогим и при разработке относительно простых систем просто экономически невыгодным. К тому же конфигурирование и настройка подобной системы могут потребовать значительного времени, особенно если решаются нестандартные или узкоспециализированные задачи;
- установить одну из промышленных специализированных систем с предопределенными функциями обработки и передачи данных для конкретной области применения. Как правило, такие системы включают изме-

рительные преобразователи и периферийные модули управления, которые управляются по интерфейсу RS-485 с центрального модуля на мощном микроконтроллере или цифровом процессоре сигналов. Центральный обрабатывающий модуль включает, как правило, сетевой интерфейс Ethernet, по которому данные передаются на удаленный компьютер. Такие системы проще и дешевле по сравнению со стандартным оборудованием для промышленных сетей, но их программирование и модификация под конкретные задачи могут стать серьезной проблемой в силу их узкой специализации и жесткой привязки к аппаратной части. Аппаратное решение может быть недорогим, но программирование и настройка периферийных модулей, включая сетевой интерфейс, может занять достаточно много времени и потребовать изучения массы технической документации;

- применить недорогие стандартные измерительные модули на базе хорошо изученных и легко программируемых микроконтроллеров серий 8051/8052 или PIC, которые будут передавать предварительно обработанные данные по интерфейсу RS-232/RS-485 на компьютерную систему с работающей операционной системой Windows. Полученные по последовательному каналу данные можно передать по сети, без особого труда разработав сетевое приложение с помощью одного из популярных инструментальных средств, таких, например, как Microsoft Visual Studio .NET или Delphi. Для разработки такой системы как нельзя лучше подойдут рассмотренные нами в предыдущем разделе виртуальные порты, а сетевую часть приложения клиента и сервера можно проверить на одном компьютере. Таким образом, большую часть программного обеспечения можно разработать и проверить на одном и том же ПК без какого-либо дополнительного оборудования. Модифицировать такие системы под другие задачи несложно, и это не потребует много времени.

Рассмотрим реализацию подобного проекта, промоделировав функционирование интерфейса RS-232 и передачи данных по сети TCP. В качестве внешнего устройства будет выступать программа-эмулятор терминала, передающая текстовые данные через последовательный виртуальный порт COM2. Данные будут приниматься тестовым приложением, выступающим в роли Web-сервера, на виртуальном порту COM5.

Web-сервер ожидает входящих подключений приложений-клиентов на порту 8089 и передает им полученную с порта COM5 строку. Просмотреть данные, полученные через последовательный порт, можно, подключившись к Web-серверу посредством браузера (browser), например, Internet Explorer 7. Все

это можно выполнить на одном компьютере, если в качестве IP-адреса использовать 127.0.0.1.

Исходный текст приложения-сервера показан в листинге 6.4.

Листинг 6.4. Приложение Web-сервера

```
#include <stdio.h>
#include <windows.h>

#define port 8089

BOOL fSuccess;
char *pcComPort = "COM5";
DCB dcb;

HANDLE hThread, hCom;
DWORD dwParam, dwThreadId;

char str[1024];
char com5Buf[4096];
char *pbuf;

char *str_start = "<html><body><h1> ";
char *str_end = " </h1></body></html>";
char buf[4096];
DWORD bytesRead;

VOID WINAPI ThreadProc(LPVOID* dummy)
{
   while (true)
    {
     bytesRead = ReadFile(hCom,
                          pbuf++,
                          1,
                          &bytesRead,
                          NULL);
```

```
    }
    CloseHandle(hCom);
}

void main(void)
{
  WSADATA wsd;
  SOCKET sListen, sClient;
  struct sockaddr_in server;
  int bytes;

  if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
   {
    printf("WSAStartup error: %d\n", GetLastError());
    return;
   }

  pbuf = com5Buf;
  memset(com5Buf, '\0', sizeof(buf));
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  server.sin_family = AF_INET;
  server.sin_port = htons(port);

  sListen = socket(AF_INET, SOCK_STREAM, 0);
  bind(sListen, (struct sockaddr*)&server, sizeof(server));
  listen(sListen, 5);

  hCom = CreateFile(pcComPort,
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);
  if (hCom == INVALID_HANDLE_VALUE)
```

```
    {
     printf("COM5 opening error!\n");
     return;
    }

GetCommState(hCom, &dcb);
dcb.BaudRate = CBR_9600;
dcb.ByteSize = 8;
dcb.Parity = NOPARITY;
dcb.StopBits = ONESTOPBIT;

fSuccess = SetCommState(hCom, &dcb);
if (!fSuccess)
  {
    printf ("SetCommState failed with error %d.\n",
            GetLastError());
    return;
  }

printf ("Serial port %s successfully reconfigured.\n",
        pcComPort);
GetCommState(hCom, &dcb);
printf("COM4 baud rate is %d\n", dcb.BaudRate);
hThread = CreateThread(NULL,
          0,
         (LPTHREAD_START_ROUTINE)ThreadProc,
          &dwParam,
          0,
          &dwThreadId);
if (hThread == NULL)
 {
  printf("Can't create thread: %d\n",
          GetLastError());
  closesocket(sListen);
  return;
 }
```

```
  printf("WEB-Server waits on port 8089\n");
  while(true)
   {
    sClient = accept(sListen, 0, 0);
    if (sClient != INVALID_SOCKET)
    {
      bytes = recv(sClient, buf, sizeof(buf), 0);
      if (bytes > 0)
       {
        strcpy(str, str_start);
        strcat(str, com5Buf);
        strcat(str, str_end);
        send(sClient, str, strlen(str), 0);
       }
     closesocket(sClient);
     memset(com5Buf, '\0', sizeof(buf));
     pbuf = com5Buf;
    }
  }
  closesocket(sListen);
  return;
}
```

В этом приложении основной поток выполняет функции прослушивания входящих соединений на сокете `sListen`. При поступлении запроса от клиента создается рабочий сокет `sClient`, через который выполняется обмен данными с клиентом. При запросе клиенту передается буфер данных `str`, содержащий текстовую строку, принятую с виртуального последовательного порта COM5.

Прием данных с COM5 выполняется в отдельном потоке, функция `ThreadProc` которого считывает байты данных в буфер `com5Buf`. Чтение данных в функции потока выполняет функция Win API `ReadFile`.

Приложением-клиентом, получающим данные, будет выступать стандартный браузер Internet Explorer 7, поэтому перед отправкой данных браузеру в Web-

сервере необходимо сформировать html-страницу, что выполняется с помощью следующей группы операторов:

```
strcpy(str, str_start);
strcat(str, com5Buf);
strcat(str, str_end);
```

Для тестирования запустим наше приложение Web-сервер (рис. 6.10).

Теперь запустим программу-эмулятор терминала и наберем в ней текстовую строку (рис. 6.11).

Рис. 6.10. Окно запущенного приложения Web-сервера

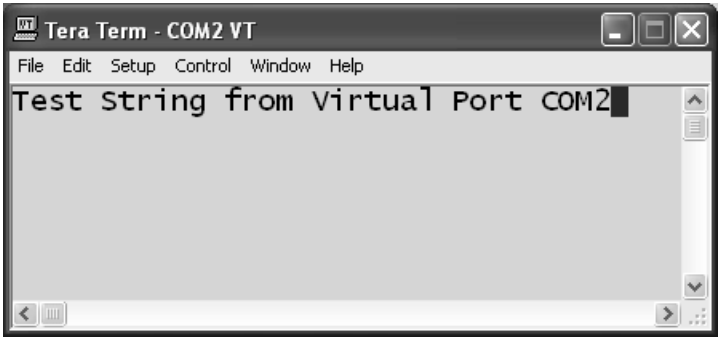


Рис. 6.11. Строка, переданная Web-серверу

Для просмотра содержимого буфера памяти на Web-сервере запустим Internet Explorer 7 и наберем следующий адрес URL:

http://127.0.0.1:8089

Web-браузер вернет строку данных (рис. 6.12).

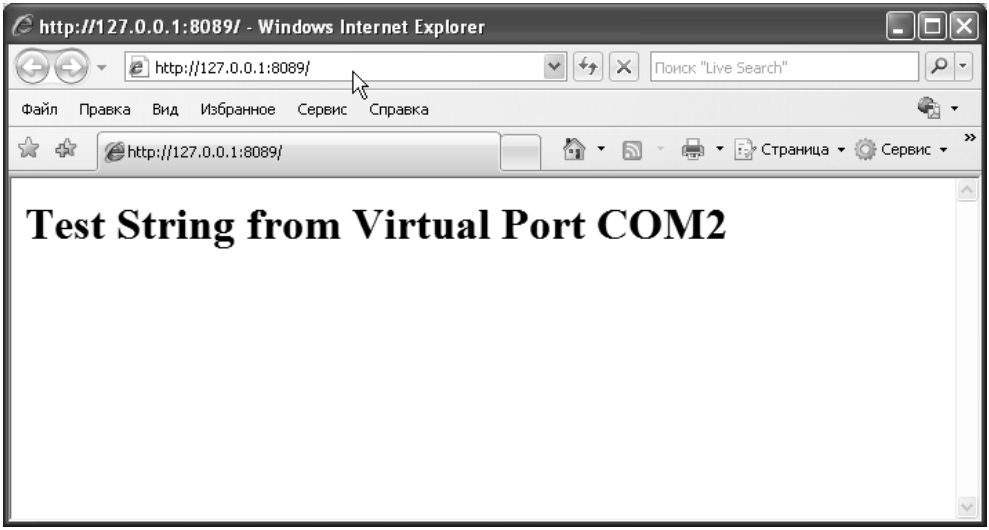


Рис. 6.12. Окно браузера при запросе данных с сервера

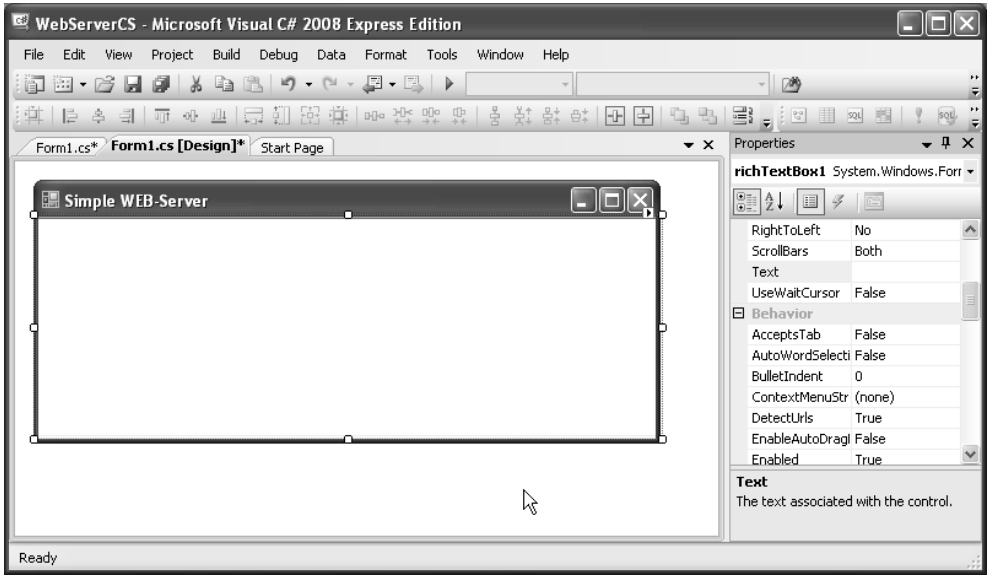


Рис. 6.13. Окно конструктора приложения

Еще один вариант реализации Web-сервера с графическим интерфейсом пользователя, передающего данные с виртуального последовательного порта COM5 по запросу интернет-клиента, рассмотрим в следующем примере.

Приложение Web-сервер разработаем в Visual C# .NET 2008. На форму приложения поместим многострочный текстовый редактор RichTextEdit, в окно которого будет выводиться строка клиентского запроса (рис. 6.13).

Большая часть функциональности сервера реализована с помощью программного кода, исходный текст которого показан в листинге 6.5.

Листинг 6.5. Исходный текст приложения Web-сервера

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.IO.Ports;
using System.Threading;
using System.Windows.Forms;

namespace WebServerCS
{
  public partial class Form1 : Form
  {
    public String buf;
    public SerialPort serialPort1;
    public static Thread myThread;
    public static TcpListener server;

    public IPAddress addr;
    public int port = 5151;
    public IPEndPoint iep;
    public int bytesRead;
```

```
 public StringBuilder sb;
 public byte[] bytes;
 public Form1()
  {
   InitializeComponent();
  }

 private void Form1_Load(object sender, EventArgs e)
 {
   bytes = new byte[1024];
   serialPort1 = new SerialPort("COM5", 9600,
               Parity.None, 8, StopBits.One);

   myThread = new Thread(ThreadProc);
   addr = IPAddress.Parse("127.0.0.1");
   iep = new IPEndPoint(addr, port);
   server = new TcpListener(iep);
   myThread.Start();
   serialPort1.Open();
}

 private void Form1_FormClosing(object sender,
                     FormClosingEventArgs e)
  {
   server.Stop();
   myThread.Abort();
   serialPort1.Close();
 }

 public void serialPort1_DataReceived
(object sender, SerialDataReceivedEventArgs e)
 {
  buf = serialPort1.ReadLine();
 }
 public void ThreadProc()
  {
   server.Start();
```

```
      serialPort1.DataReceived += new
      SerialDataReceivedEventHandler
      (serialPort1_DataReceived);
      while (true)
       {
        String sin, sout;
        TcpClient client = server.AcceptTcpClient();
        NetworkStream stream = client.GetStream();
        bytesRead = stream.Read(bytes, 0, bytes.Length);
        if (bytesRead > 0)
         {
          sin = System.Text.ASCIIEncoding.ASCII.GetString
                (bytes, 0, bytesRead);
          richTextBox1.AppendText(sin);
          sb = new StringBuilder();
          sb.Append("<html><body></h1>");
          sb.Append(buf);
          sb.Append("</h1></body></html>");
          sout = sb.ToString();
          bytesRead = System.Text.ASCIIEncoding.ASCII.GetBytes
         (sout, 0, sout.Length, bytes, 0);
          stream.Write(bytes, 0, bytesRead);
         }
        stream.Close();
        client.Close();
      }
   }
  }
}
```

В приложении будут использоваться сетевые компоненты, а также класс `SerialPort`, поэтому в начале программы нужно импортировать соответствующие пространства имен:

```
using System.Net.Sockets;
using System.IO.Ports;
using System.Threading;
```

Прием данных с виртуального порта COM5, а также обработка сетевых подключений реализованы в отдельном потоке, поэтому при запуске приложения следует инициализировать новый поток `myThread`, а также открыть последовательный порт COM5 для приема данных, что выполняет следующий фрагмент программного кода:

```
serialPort1 = new SerialPort("COM5", 9600, Parity.None,
                             8, StopBits.One);

myThread = new Thread(ThreadProc);
. . .
myThread.Start();
serialPort1.Open();
```

При создании экземпляра `serialPort1` класса `SerialPort` сразу же устанавливаются стандартные параметры обмена данными (скорость обмена 9600 бод, 8-битовая посылка, без проверки четности и с одним стоповым битом).

В функции-обработчике `Form1_Load` также создается прослушивающий сокет `server`:

```
addr = IPAddress.Parse("127.0.0.1");
iep = new IPEndPoint(addr, port);
server = new TcpListener(iep);
```

Поскольку тестирование приложений Web-сервера и сетевого клиента, в качестве которого выступает Internet Explorer 7, выполняются на одной машине, то в качестве IP-адреса сервера устанавливается значение 127.0.0.1. В качестве номера порта выберем значение 5151.

Обработка данных с виртуального порта COM5 и отправка данных клиенту осуществляется в функции потока `ThreadProc`. При запуске потока вначале стартует сокет `server` и инициализируется функция-обработчик `serialPort1_DataReceived` события `DataReceived`:

```
server.Start();
serialPort1.DataReceived += new SerialDataReceivedEventHand-
ler(serialPort1_DataReceived);
```

Затем в бесконечном цикле `while` осуществляется обработка входящих запросов на подключение к серверу. При получении такого запроса создается рабочий сокет `client`, посредством которого выполняется обмен данными с приложением-клиентом:

```
TcpClient client = server.AcceptTcpClient();
```

Для приема-передачи данных теперь можно использовать поток `stream`:

```
NetworkStream stream = client.GetStream();
```

Данные, передаваемые по сети, представляют собой поток байтов, для приема которого следует использовать метод `Read` сетевого потока `stream`:

```
bytesRead = stream.Read(bytes, 0, bytes.Length);
```

Если был принят запрос клиента (количество принятых байтов `bytesRead` отлично от нуля), то принятый поток байтов после преобразования в строку `sin` отображается в окне текстового редактора RichTextBox:

```
sin = System.Text.ASCIIEncoding.ASCII.GetString
      (bytes, 0, bytesRead);
richTextBox1.AppendText(sin);
```

Затем формируется ответ приложению-клиенту (в данном случае, это браузер Internet Explorer 7) с помощью следующего фрагмента программного кода:

```
sb = new StringBuilder();
sb.Append("<html><body></h1>");
sb.Append(buf);
sb.Append("</h1></body></html>");
sout = sb.ToString();
```

Сформированная строка ответа `sout` перед отправкой должна быть преобразована в массив байтов `bytes`, что выполняет оператор

```
bytesRead = System.Text.ASCIIEncoding.ASCII.GetBytes
            (sout, 0, sout.Length, bytes, 0);
```

Наконец, содержимое массива `bytes` передается клиенту:

```
stream.Write(bytes, 0, bytesRead);
```

И рабочий поток `stream`, и сокет `client` нам больше не нужны, поэтому можно их закрыть, что и выполняется с помощью операторов

```
stream.Close();
client.Close();
```

По завершении работы всего приложения следует закрыть виртуальный порт COM5, прослушивающий сокет `server`, и завершить поток `myThread`:

```
private void Form1_FormClosing(object sender,
                        FormClosingEventArgs e)
  {
   server.Stop();
   myThread.Abort();
   serialPort1.Close();
  }
```

Несколько слов о функции-обработчике `serialPort1_DataReceived`. В ней присутствует единственный оператор `buf = serialPort1.ReadLine();`, который принимает полученные по последовательному порту данные в буфер `buf`. Поскольку функция-обработчик запускается не в основном, а во вспомогательном потоке `myThread`, то ее функционирование будет корректным.

Для тестирования приложения запустим наш Web-сервер и программу-эмулятор терминала на виртуальном порту COM2, в окне которой наберем строку (рис. 6.14).

Рис. 6.14. Передача тестовой строки Web-серверу

После отправки строки серверу запустим Internet Explorer 7 и наберем адрес нашего Web-сервера. После этого в окне браузера будет отображена строка из приложения-терминала, а в окне Web-сервера — текст запроса (рис. 6.15 и 6.16).

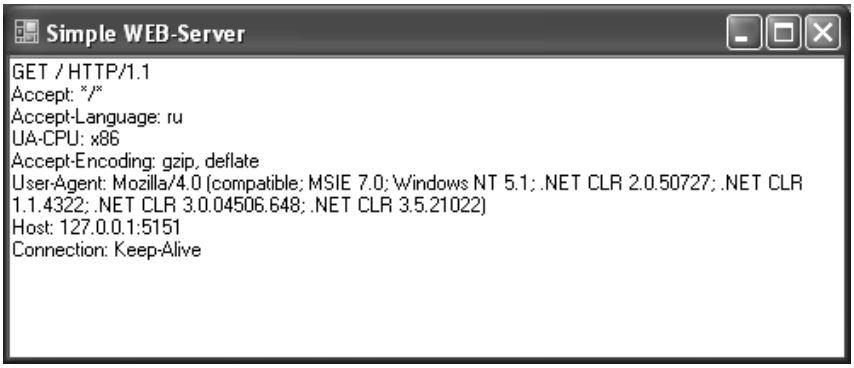


Рис. 6.15. Окно Web-сервера

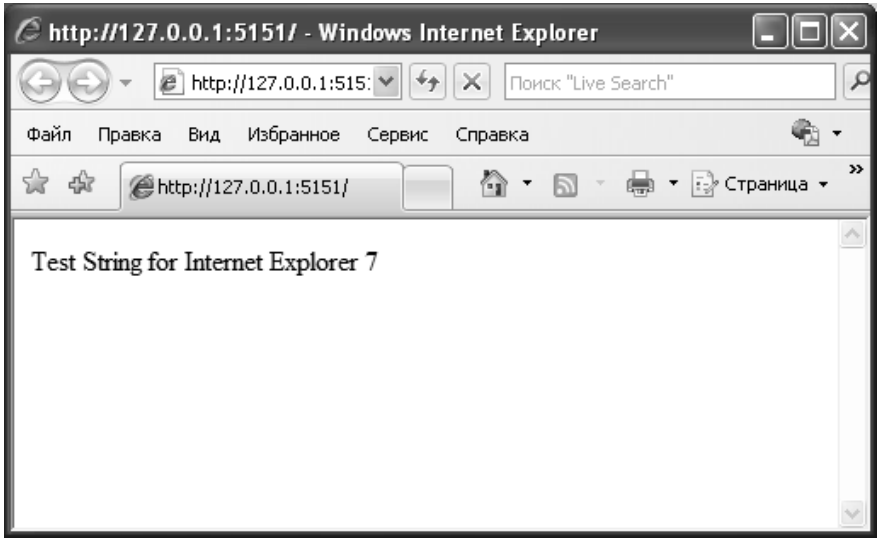


Рис. 6.16. Окно браузера

Рассмотренные ранее сетевые приложения являются демонстрационными, и на их базе можно создать собственные приложения, работающие с несколькими последовательными портами по сети TCP/IP. При этом нужно предусмотреть обработку исключительных ситуаций, которые могут возникнуть при работе реальных систем сбора и обработки данных.

К стандартным сбоям можно отнести потерю сетевого соединения, неисправность интерфейса RS-232/RS-485 и т. д. Все подобные ситуации должны быть учтены при разработке программного кода.

Многие фирмы-производители программного обеспечения выпускают приложения, подобные рассмотренным, для подключения последовательных портов к сетям Интернета и обработки данных.

В качестве одного из примеров таких приложений приведу программу *Serial to Ethernet Connector* фирмы Eltima Software GmbH. Программа обеспечивает удобный способ для установки соединения "клиент-сервер" между двумя COM-портами по сети Ethernet. Вид окна работающего приложения показан на рис. 6.17.

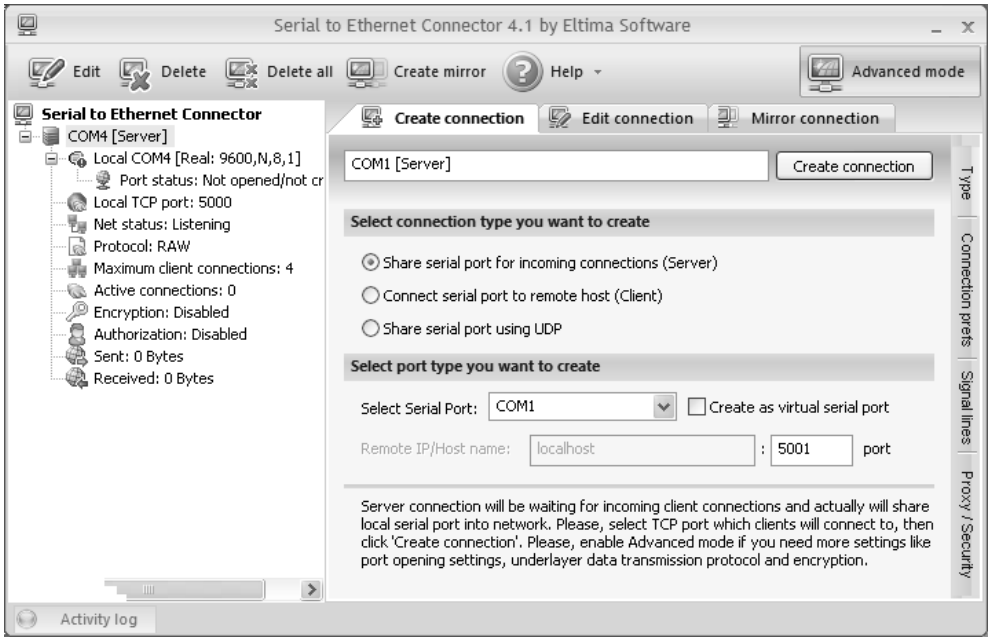


Рис. 6.17. Вид окна работающего приложения

Сервер допускает до 255 клиентских соединений и позволяет работать как с реальными (физическими), так и с виртуальными COM-портами.

Рассмотренные нами примеры приложений работают с интерфейсом RS-232, однако их можно использовать и для обмена данными по RS-485, если выполнить определенные аппаратные доработки интерфейса RS-232. В этой книге анализировалась аппаратная реализация протокола RS-485 на микросхеме MAX487, которую можно использовать и с рассмотренными ранее проектами.

Заключение

Последовательные интерфейсы передачи данных RS-232 и RS-485 в настоящее время используются во многих промышленных и лабораторных системах сбора и обработки информации. Основной недостаток ранних систем на базе RS-232 — низкая скорость обмена — в настоящее время уже не является актуальным с учетом разработки все более скоростных приемопередатчиков. Протокол RS-232 в обозримом будущем и далее будет широко применяться в системах передачи данных, в которых требуется высокая надежность и простота реализации, а его продвижение в область более высоких скоростей только укрепит позиции этого стандарта на рынке.

Приложение

Описание компакт-диска

На прилагаемом к книге компакт-диске содержатся файлы с исходными текстами программ, приведенных и описанных в книге. Файлы размещаются в каталогах Ch0x, где x — номер главы (2, …, 6). В наименованиях файлов первая слева цифра указывает на номер главы, а вторая, следующая после нижнего подчеркивания цифра, на соответствующий номер листинга в этой главе. Например, файл Листинг_2_10 приведен и рассмотрен в листинге 2.10 *главы 2*. Для компиляции и сборки программы нужно включить соответствующий файл в разрабатываемый проект в среде Visual Studio .NET, Delphi, Keil C51 (для микроконтроллеров 8051/8052) или Microchip MPLAB C30 (для микроконтроллеров PIC24).

Предметный указатель

П

С

У

Ф